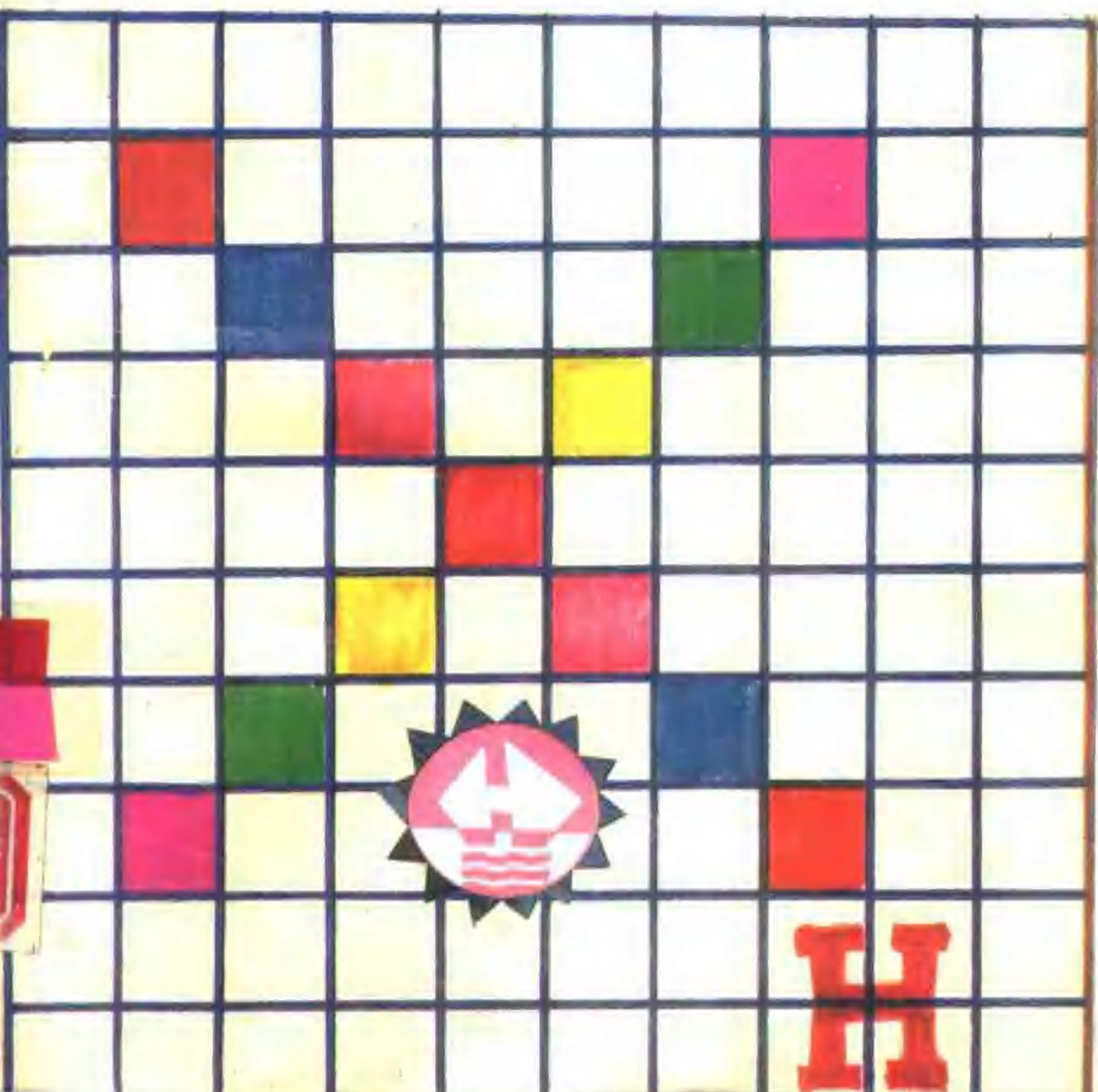


HOPE COMPUTER COMPANY LTD.

适用于IBM—PC 机 长城0520机 WANG—PC 机

PASCAL 程序设计 基础及实用技术

王 诚 赵毓升 奚和泉 编译



北京希望电脑公司

IBM-PC机 长城0520机 WANG-PC机

PASCAL程序设计基础 及实用技术

王 诚 赵毓升 奚和泉 编著

北京希望电脑公司

一九九二年四月

封面设计：京 艳

■ 北京市新闻出版局

准印证号： 891167

■ 订购单位：北京 8721 信箱资料部

■ 电 话： 2562329

■ 电 传： 01—2561057

■ 电 挂： 0755

■ 地 址：海淀影剧院北侧

■ 乘 车： 320、332、302路海淀黄庄下车

■ 办公地点：公司大楼 101 房间

前 言

目前我国已有各类微机数十万台，其中功能比较齐全、用得又比较普及的还是 IBM PC 机、长城 0520 机和 WANG PC 机，它们占已有微机总数的绝大部分，今后每年仍会以数万台的增长速度在发展。

从这些微机的应用领域和使用水平看，总的状况是令人鼓舞的，但也有相当多的使用单位遇到困难。造成这一状况的根本原因，是使用单位的人员得不到必要的培训，自学又受到种种条件，特别是缺乏水平适中、简洁实用的技术资料的限制，在计算机能做些什么事情和怎样使用计算机两个方面缺乏基本和必要的知识。

要解决这一矛盾，我们认为至少应做好如下两项技术方面的工作：

第一，要大力普及一种结构性能良好，语句简明清晰，数据类型完备，易学、好用且运行效率高的程序设计语言，解决能使用计算机的一个关键问题。我们推荐用 PASCAL 语言，因为它完全满足上述要求，正如我国计算机专家金兰教授所说的：“应该创造条件逐步用 PASCAL 语言取代 BASIC 语言在我国目前占据的不适当地位，要把 PASCAL 语言作为推广计算机应用的启蒙语言，让人们从一开始就培养严谨的、良好的程序设计风格 and 习惯，然后再学习 FORTRAN、COBOL 等带有一定应用偏向的语言。这样做将对人们的程序设计基本训练产生深远的影响。”

第二，要开发出一批能在微机上使用的完全汉字化的新型高性能的关系数据库系统，使其最大限度地符合我国国情和目前广大用户的技术素质与水平。再进一步，最好使这些数据库系统带有一定的开发工具或自生成能力，使一般水平的用户能用这个系统实现自己单位的业务、事务管理，这是促使各单位的计算机真正运转起来并带来效益的重要条件。是让广大用户了解计算机能做些什么事情的最好办法。

基于上述分析，清华大学计算机系的一个科研课题组，在多年教学和科研工作的基础上又经过两年多的艰苦工作，已基本完成在上述机种上运行的一个数据库系统和管理信息系统的开发工具系统。这套软件是以 PASCAL 语言为主并配以少量汇编语言子程序实现的。

《PASCAL 程序设计基础和实用技术》这一资料，是在多年教学实践和科研实践的基础上编写的。我们期望它能对扩大微机应用的范围和使用水平有所贡献，同时也希望它成为将来使用、进一步开发完善我们的软件系统的用户的基础读物。

这本资料主要由两部分内容组成：

第一部分，简明、准确地讲解 PASCAL 程序设计的基础知识。在全面介绍标准 PASCAL 功能的基础上，重点突出了 MS PASCAL 语言扩展功能的用法，针对性很强。这部分内容是以教学讲义形式编写的，循序渐进，由浅入深，并配有很多原理性的小程序，对大多数微机用户和工程技术人员来说，比较容易学懂会用。

第二部分，详尽、分类地介绍了 PASCAL 程序设计中的许多实用技术。这些技术的提出和实现方法，多是以实现数据库系统为例进行讲解的，但必须指出，这是一些通用技术，在其它各应用领域也都有普遍实用意义。这一部分内容涉及面广，难度也比较大，但

非常实用，是在其它书籍资料中难以找到的。

因为本资料的内容是专门针对 IBM PC 机、长城 0520 机和 WANG PC 机编写的，针对性强，新素材多，简明实用，所以对直接使用这些机种的人员更对口。对不使用这些机种的人员来说，不必过多阅读 MS PASCAL 扩展功能的部分。对第二部分中给出的内容，可以看其思路和方法，不必仔细阅读程序。

本资料也可以作为听《PASCAL 程序设计》课并在 IBM PC 机上上机的学生的教材。此时，应先学标准 PASCAL 的功能，再学 MS PASCAL 的部分扩展功能，最后以练习或演示方式学习第二部分中的内容。

目 录

前言	1
第一章 PASCAL语言概述	3
1.1 PASCAL语言的特点和用途	3
1.2 运行MS PASCAL程序的方法和步骤	4
1.3 MS PASCAL语言的特性	6
第二章 简单数据类型和简单程序设计	12
2.1 整型数据及其运算	43
2.2 字类型、字节类型数据及其运算	45
2.3 实型数据及其运算	46
2.4 PASCAL语言中的算术运算	48
2.5 算术运算程序设计举例	49
2.6 字符类型数据及其运算	65
2.7 布尔类型数据及其运算	66
2.8 枚举类型数据及其运算	66
2.9 子界类型数据及其运算	68
2.10 标准函数及其引用	69
2.11 关系运算和逻辑运算	71
2.12 过程类型	72
2.13 简单程序设计举例	73
第三章 程序设计中的流程控制	78
3.1 顺序结构程序设计	78
3.2 分支结构程序设计	79
3.3 循环结构程序设计	87
3.4 MS PASCAL语言在分支与循环控制结构中实现的扩展功能	91
3.5 循环结构程序设计举例	93
第四章 构造型数据类型	100
4.1 数组类型	101
(高级数组类型, 字符串类型, 变长字符串类型)	
4.2 记录类型和WITH (开域) 语句	109
4.3 集合类型	115
4.4 文件类型	121
4.5 地址类型数据及其引用	139
4.6 指针类型	149
4.7 相同类型和兼容类型问题 (• 有扩展内容)	158
第五章 过程和函数	161
5.1 可用的过程和函数	161

5.2	过程说明和过程调用	174
5.3	函数说明和函数引用	176
5.4	过程和函数的参数类型	176
5.5	程序中的模块与层次结构设计问题	182
5.6	程序设计举例	186
第六章	屏幕上格式数据的输入输出支持	212
6.1	终端硬件特性及其使用方法	212
6.2	实现屏幕格式数据输入输出的基本技术	232
6.3	在用户程序中使用支持软件	240
6.4	在屏幕上实现多窗口入出的技术	244
6.5	屏幕页的概念与用法	248
第七章	制表软件和报表支持功能	255
7.1	制表软件实现的基础	255
7.2	制表软件的进一步完善	261
7.3	屏幕编辑方式制表	268
7.4	程序中直接形成表格	275
7.5	用表格方式从终端录入数据	280
7.6	报表生成及打印	284
7.7	实现绘图功能	289
7.8	打印机的打印控制	299
第八章	索引文件的实现及应用	310
8.1	索引文件的实现原理	310
8.2	支持索引表文件操作的程序	314
8.3	在用户的程序中使用索引文件	331
8.4	实现与使用索引文件中的实际问题	338
第九章	和汇编语言程序的接口关系	339
9.1	问题的提出	339
9.2	汇编语言程序设计的简明知识	339
9.3	在 PASCAL 程序中调用汇编语言的子程序	349
9.4	实现音响控制和仿真电子琴的程序	354
9.5	协同运行多个程序和使用公用数据区	357
9.6	在程序中查询系统信息	370
第十章	文件和磁盘的应用技术	378
10.1	文件控制块 (FCB) 和文件用法	378
10.2	磁盘的盘区分配及读写	389
10.3	文件目录的直接操作	395
10.4	利用串行口实现计算机之间的通信	399
10.5	在 PASCAL 程序中读写 dBASE 的文件	404
附录 A	ASCII 字符集	411
附录 B	MS PASCAL 语言的语法图	412
附录 C	出错信息一览表	422

第一章 PASCAL语言概述

1.1 PASCAL语言的特点和用途

一个计算机系统由硬件系统和软件系统两部分组成。硬件系统是计算机系统在实际装置的总称,软件系统通常是指计算机系统运行所需的各种程序及其有关数据。

计算机硬件系统一般包含中央处理机(通称CPU)、内存储器、外存储器(如磁盘机、磁带机)以及输入和输出设备(如终端、行式打印机等)。

计算机软件系统通常包括操作系统、系统实用程序(如编辑程序,装配程序,调试程序)、语言处理程序(如汇编程序、编译程序、解释程序等)以及某些专用程序。

近来,许多计算机系统中,一般还配有不同功能的数据库管理系统、网络通讯系统等软件。当然,在一个实用的计算机系统中,往往还有应用程序(用户程序)这一类软件。

计算机程序设计语言是用来编写计算机程序的工具。可以从不同的角度对程序设计语言进行分类。

根据语言提供功能的水平,可以把语言划分为低级语言和高级语言。低级语言一般指汇编语言,多数是面向具体的计算机的,较难掌握与使用。但用它设计的程序,能直接使用计算机系统中全部的硬件成分与功能;而高级语言一般是面向应用的,与具体的计算机硬件关系不那么密切,亦较标准与规范,且容易掌握与使用。如FORTRAN, BASIC, PASCAL, COBOL, PL/1以及C语言等都属于高级语言。

按照语言的应用范围,可以把语言划分为通用语言和专用语言。通用语言可适用于多种应用领域,如实现各种科学计算、不同部门的事务管理以及研制计算机的系统软件等。上述几种高级语言,还有LISP和PROLOG语言等都属于流行的通用语言。专用语言是指特定用于某一领域的程序设计语言。用在不同领域中的专用语言名目繁多,其专用性限制了它们的应用范围。因此,了解与使用专用语言的人数相对较少。

PASCAL语言,属于通用的高级程序设计语言。由于它完全符合结构程序设计的原则与要求,又具有语句简明灵活,表达能力强,数据类型丰富完备、用户可按需要自行定义等特点,使得它在科学计算、事务管理、系统软件设计和高等学校多门课程的计算机教学等领域得到越来越广泛的应用。再加上PASCAL语言的程序书写格式自由、程序设计的风度优美、可读性好、编译紧凑方便、目标程序运行效率高等特点,正日益受到人们的重视。

关于PASCAL语言的具体特征,我们不妨把PASCAL语言的设计者沃斯教授的一段话摘录在下面,供大家参考。

“我们把PASCAL的特征列举如下:

- (1) 变量说明是强制性的。
- (2) 某些基本字(例如, BEGIN, END, REPEAT等)是‘保留’的,不能作标识符。
- (3) 分号(;)看作是语句分隔符,而不是语句终止符(如同PL/1语言一样)。
- (4) 标准数据类型是整数、实数、字符和布尔类型。基本的数据构造工具有数组、记录(对应于

COBOL 和 PL/1 语言的‘结构’), 集合和文件。这些结构可通过组合和嵌套以形成集合的数组、记录的文件, 等等。数据还可以动态地分配并由指针访问。

(5) 集合 SET 数据结构提供了类似于 PL/1 语言的‘位串’的能力。

(6) 数组可以具有任意维数与任意的界, 数组的界是常量, 即没有动态数组。

(7) 象 FORTRAN、ALGOL 和 PL/1 语言一样, 有转移语句。标号是无符号整数, 必须加以说明。

(8) 复合语句如同 ALGOL 语言, 相应于 PL/1 语言里的 DO 群。

(9) ALGOL 语言的开关和 FORTRAN 语言的计算转语句的功能在 PASCAL 中由情况语句表示。

(10) 循环语句相应于 FORTRAN 的 DO 循环, 它的步长只能是 1 (TO) 或 -1 (DOWNT0), 并仅当控制变量的值落在界内时执行。因此, 被控制的语句可能完全没有执行。

(11) 没有条件表达式和多重赋值。

(12) 过程和函数可以递归调用。

(13) 对变量不存在 (象 ALGOL 语言里那样) ‘OWN’ 属性。

(14) 参数有值调用和引用调用, 没有名字调用。

(15) 就不存在匿名的分程序来说, 这里的‘分程序结构’不同于 ALGOL 和 PL/1 语言的分程序结构, 即每个分程序都给一个名字, 并做成一个过程。

(16) 常量、变量等所有对象必须在它们被引用之前加以说明。但允许以下两个例外,

- 指针类型定义里的类型标识符;
- 当存在向前引用时的过程与函数调用。

许多人初次接触 PASCAL 语言时, 往往感到它缺少某些‘中意的特色’——例如方幂运算符、串的并列、动态数组、布尔值的算术运算、自动的类型转换和省缺说明等, 这都不是疏忽遗漏, 而是有意剔除的。在某些场合下, 有了它们反而会招致程序设计解法上的效率不高问题; 在另外一些场合, 又会感到它们不利于达到程序设计清晰可靠和‘风度优美’的目标。最后, 还得考虑对繁多的有效的程序设计功能进行严格挑选, 以保证编译程序比较紧凑和高效。”

1.2 运行 MS PASCAL 程序的方法和步骤

为了运行 MS PASCAL 的程序, 要求个人计算机的配置如下,

- 个人计算机主机 (CPU 和内存)。
- 终端。
- 双软盘驱动器, 或一个软盘驱动器和一个温氏硬盘。
- 最好再有一台打印机。
- 几片 5 1/4 英寸的软磁盘片。

• 除了有操作系统等系统软件外, 当然还必须有 PASCAL 语言的编译程序, 它由以下几个文件组成: (WANG PC 机)

——PAS1.EXE
——PASKEY
——FILKQQ.INC
——FILUQQ.INC
——PAS2.EXE
——PASCAL.LIB

——PASCAL

在有些机器中,有时还有另外一些文件。

通常情况下,PASCAL的编译程序是以软盘片形式提供给用户的。我们建议用户,最好不要直接使用这几块软盘片去工作,而是首先拷出一份备份来。工作时使用拷贝出来的几块盘片,而把原版盘片保存好,以备由于误操作或正常磨损造成工作盘上的PASCAL编译程序出现问题时,能再进行软件复制用。在系统中配有硬盘的情况下,可把PASCAL的编译程序直接拷到硬盘上。

运行一个MS PASCAL的程序,通常的步骤及其含义如下:

1. 用个人计算机上的编辑程序,建立MS PASCAL语言的源程序,或从已存放在软盘上的文件中拷贝。按DOS操作系统规定,文件名应由1到8个字符组成,扩展名通常应由3个字符组成。任何一个PASCAL源程序文件的扩展名都应为'.PAS'。

2. 用PAS1.EXE对PASCAL源程序进行编译,将得到两个目标文件,其扩展名分别为'.SYM'和'.BIN',分别存放符号表和中间二进制代码的内容。按用户要求,还可以得到一个编译清单文件,扩展名为'.LST'。在编译的过程中,如果发现用户源程序中有错误,编译程序还会把出错情况显示在终端屏幕上。需要时,返回第一步重新编辑用户源程序。

3. 在PAS1.EXE成功地编译了用户的源程序之后,再用PAS2.EXE进行代码生成和代码优化,以便得到用户程序的二进制代码的、浮动地址形式的目标文件。如果PAS1.EXE发现用户源程序中有某些错误,它就不生成符号表和中间代码两个文件,此时PAS2.EXE将不能运行。正常情况下,PAS2.EXE都能正确运行,不会在用户程序中发现新的错误。

4. 用连接程序LINK.EXE建立用户程序的最终运行文件。它要把用户程序、PASCAL.LIB和用户的其它程序块连接成一个程序,在DOS操作系统支持下,能直接装入内存并加以运行。

5. 运行得到最终的目标程序。若发现运行中出现问题或结果不正确时,要到自己的源程序中查明问题,并返回第1步,用编辑程序改正源程序中的错误,并重复第2—第5步,直至得到满意的结果。必须说明,最终目标程序可以被反复运行,不是每运行一次都要重复第1或第2到第5步这个全过程的。

例如,假定用户的源程序名为myfile.PAS,上述五个步骤可如下进行(↵代表按RETURN回车键),

EDIT myfile.PAS↵ (新建或修改用户的PASCAL源程序)

PAS1 myfile↵

Object filename[MYFILE.OBJ]:↵

Source listing[NUL.LST]:myfile↵

Object listing[NUL.COD]:↵

(带下横线部分的内容是PAS1自动给出的提示信息,其它内容是用户从终端上打入的。编译时,用户总得给出被编译的源程序的文件名,这里为myfile,它的扩展名部分.PAS可省略。目标文件通常是需要的,PAS1在方括号中提示该文件名为MYFILE.OBJ,若用户无特殊要求时,可直接按一次回车键回答,以表示认可。源程序的编译清单文件的文件名,PAS1提示的为NUL.LST,当用户直接用回车键回答此项提示时,表示不要求给出此文件;若要求PAS1编译过程中产生此文件,则必须给出一个文件名(如这里的myfile),然后按回车,就会得到myfile.LST文件。最后的一项提示,是目标

代码的反汇编清单文件，其扩展名为 'COD'。绝大多数程序设计人员都不使用此文件，因此可直接用回车回答此项提示，那么就不产生此文件。而当用户想取消这三项提示，直接使用 PAS1 在方括号中提示的文件名时，可以用 PAS1 myfile; ✓ 的方式调用编译程序，即在文件名后打入一个分号后再回车。

PAS 2; ✓

(打入 PAS 2 后直接回车是正常用法)

LINK myfile; ✓ 或

LINK myfile; ✓ 或

LINK; ✓

(第一种方式将取消 LINK 给出的全部提示信息，直接开始连接操作。第二种和第三种方式，用户必须回答 LINK 的每一项提示)，例如：

Object Modules: myfile; ✓

Run File: myfile; ✓

List File: [MYFILE.MAP]: ✓

Libraries []: ✓

Publics [No]: ✓

Line Numbers [No]: ✓

Stack Size [Object file stack]: ✓

Load Low [Yes]: ✓

DSAllocation [No]: ✓

(对这些内容有兴趣的读者请查阅 DOS 手册中关于 LINK 的用法一章，我们在这里不再详细介绍通常情况下用第一种方式即可。如有多个 OBJ 类型的文件要连接到一起，可用空格或 '+' 号把多个文件名隔开)

myfile; ✓

(运行得到的最终目标程序)

1.3 MS PASCAL 语言的特性

一、概 述

在微机上运行的 PASCAL 语言有几个性能各异的编译程序，它们是由不同的软件公司设计的。我们这里讲的 MS PASCAL 语言的编译程序是由美国的 MICROSOFT 公司设计的，现在已有了几个版本，其功能是高版本的强，并且能兼容低版本的程序。

MS PASCAL 总的来说是符合 ISO (国际标准化组织) 草案 (ISO/TC 97/SC5 N395) 的，其目的是允许在微机上不加修改地编译并运行标准 PASCAL 语言的程序。

MS PASCAL 语言又扩充了许多新的功能，主要包括以下几部分：

- 编译命令
- 属性
- 字符串
- 系统中实现的扩展
- 单元 (UNIT)
- 高级数组类型
- 常量值设置

1. 编译命令 (COMPILER DIRECTIVES)

MS PASCAL提供了许多条编译命令。它们的作用是为用户提供各种控制、指示编译程序如何运行的手段,使其满足用户的不同要求。包括对出错检查码的控制,列表文件格式控制,条件编译的结构控制,把其他源文件穿插到被编译的源文件中的控制等等。编译命令又称作“元语言”命令 (METALANGUAGE COMMANDS, 或 META COMMANDS)。

2. 单元 (UNIT)

MS PASCAL提供了把一个很大的程序分解为多个程序单元,分别设计、编译,然后连接到一起的结构化性能很好的程序设计手段,这就是UNIT。UNIT就是可以分别设计、编译并可以方便地连接到主程序中的程序单元。比起使用外部过程来,它具有更好的结构化特性,更强的功能和更大的灵活性。一个UNIT通常由接口 (INTERFACE) 和实现 (IMPLEMENTATION) 两部分组成。

3. 属性 (ATTRIBUTES)

变量,过程和函数的属性给用户提供了对文件连接的控制手段。其中包括:

- 供全程标识符连接用的 PUBLIC和EXTERN。
- 供变量用的STATIC 和 REANONLY。
- 供过程和函数用的PURE。

4. 高级数组 (SUPER ARRAY)

MS PASCAL的高级数组类型,指的是要求给出数组下标的下界值,而不具体给出下标的上界值的变长数组。

高级数组类型不能用来在变量说明部分直接说明变量,它的主要用法是:

- 用作过程和函数的变量形式参数
- 用作指针的引用类型
- 用于“派生”出新的数组类型

5. 字符串 (STRING 和 LSTRING)

在标准PASCAL语言中,字符串变量的长度是不能变的。MS PASCAL则又提供了说明与使用变长字符串的能力。变长字符串是一种抽象的数据类型,其定义为:

SUPER PACKED ARRAY [0..*] OF CHAR, 这个串的0号单元用于记忆该串的实际长度。MS PASCAL还提供了一组对字符串进行各种操作的过程和函数。串变量的赋值、比较和读写操作都可以直接进行。

MS PASCAL提供了STRING和LSTRING两种字符串变量,前者的下标上界最大为MAXINT,后者的下标上界最大为255,在用法上它们也有一些差别。

6. 常量值设置

MS PASCAL提供了设置常量值的功能。常量值设置是在编译过程中完成的。其中包括:

- 用已定义过的某些常量定义一个新的常量。
- 给出了串常量连接操作符,还允许对数组和记录常量进行运算。
- 在程序中增设了VALUE段,为有关变量设置初值。
- 过程和函数的形式参数可以用CONST作前缀,其效果与用VAR作为前缀类似,差别在于对应的实际参数可以是一个常量,也可以是变量,并且不能在过程和函数执行体内

修改它的值。

• 数值可以用二进制、八进制、十进制或十六进制表示。在READ和WRITE过程中也支持这种表示方法。

7. 系统中实现的扩展

下面给出的扩展功能，是与使用的计算机系统的硬件、系统软件有直接联系的。用了这些扩展功能的PASCAL程序难于移植到其它计算机系统中。这些扩展是：

- 字 (WORD) 类型
- 地址 (ADDRESS) 类型
- 输入/输出能力
- 交互式的READ
- 随机文件和文件方式
- 内部过程和函数

MS PASCAL 的这些扩展功能，大大增强了语言的表述能力，使得程序设计变得更容易。当然，要正确掌握这部分内容，必须花费一些时间和精力。对于要经常使用微机的人来说，学好这部分内容是非常有益的。

二、字符集和字汇表

在一个语言中能使用的全部字符构成该语言的字符集。一种计算机语言往往还要使用某些英文单词或它们的缩写形式，这些单词构成该语言的字汇表。MS PASCAL 使用的英文单词分为两类，一类是单词含义确定，用户不能变更的，我们称它们为保留字。另一类是单词含义已定，但用户还可以重新加以确定的，我们称它们为预先说明的标识符。我们明确告诫读者，最好不把预先说明的标识符重新定义后当作用户自己的标识符用。为了叙述和学习上更简明清晰，我们将把MS PASCAL划分为四个层次：元语言、标准PASCAL、扩展PASCAL和系统PASCAL。

(1) 元语言

元语言是用以设置编译程序的可选项和进行条件编译的，由全部的编译命令组成。

(2) 标准PASCAL

MS PASCAL提供标准PASCAL语言的全部功能，使所有标准的ISO PASCAL程序都能正确地进行编译和运行。

(3) 扩展PASCAL

MS PASCAL提供了相对“安全”的扩展功能，从而使ISO PASCAL的功能得到增强，因此程序可以设计得更好。这些扩展包含象BREAK语句，UNIT和结构常量那样的结构等等。

(4) 系统PASCAL

MS PASCAL还提供了某些更完善的扩展，包括那些对系统程序设计有用的或必要的扩展。这些附加的扩展包括地址类型和RETYPE函数等等。

1. 字符集

基本的PASCAL词汇是由字母、数字、保留字和专用字符组成的。小写和大写字母可以互换，仅仅在字符串文字中不能这样做，是一个例外。

专用字符按照如下的类别分成组，

- 元语言

\$ 只用作元语言的前缀。

- 标准PASCAL

+ - * / = < > () [] . , ; ' ` } { ^ < > > = < = : =。在标识符中允许使用 “_” (下划线)。

- 替换

可用 (*...*) 代替 {...}; 可用 (, 代替 [; 可用 .) 代替]; ? 或 @ 可用来代替 ^, 以代表一个指针。

- 更高级的替换

“#” 用在整数常量中 (表示是16进制的数)。“!” 表示行的结束。用 [...] 代替 BEGIN...END。

- 无用的字符 % & ? | ~ , \

空格 (空白) 符、制表符和换页符也是标准字符集的一部分。制表符 CHR (9) 看作是空格符并可以传送到清单文件上。换页符 CHR(12) 被转换成新页元命令 (\$PAGE+)。

包括在 CHR (0) 到 CHR (31) 中的所有其它字符, 上面列出的那些无用的字符, 和从 CHR (127) 到 CHR (255) 这些字符用在源文件中时, 就会产生错误。仅当它们出现在注释或字符串文字中的时候才是允许的。

(, 和] 配对或 [和 .) 配对是允许的 (但这里不推荐它)。{ 必须同 } 配对, (* 必须和 *) 配对。

2. 字汇表

PASCAL保留字

下面这些字在标准PASCAL中是保留的,

AND	END	NIL	SET
ARRAY	FILE	NOT	THEN
BEGIN	FOR	OF	TO
CASE	FUNCTION	OR	TYPE
CONST	GOTO	PACKED	UNTIL
DIV	IF	PROCEDURE	VAR
DO	IN	PROGRAM	WHILE
DOWNT0	LABEL	RECORD	WITH
ELSE	MOD	REPEAT	

MS PASCAL的特性又增加了下列一些保留字,

特性 (FEATURE)

单元接口

模块

扩展CASE

保留字

IMPLEMENTATION
INTERFACE
UNIT
USES
MODULE
OTHERWISE

高级数组类型

控制流程

扩展运算符

地址类型

数值段

属性 (ATTRIBUTES)

“属性”是用来表示变量、过程或函数的特性的关键字。下列的一些属性是保留字:

EXTERN	PUBLIC	READONLY
EXTERNAL	PURE	STATIC

命令 (DIRECTIVES)

“命令”是用于过程或函数块中的字。

EXTERN	EXTERNAL	FORWARD
--------	----------	---------

注意: EXTERN 既是属性又是命令。EXTERNAL 是 EXTERN 的同义词, 它具有同其他几个 PASCAL 兼容的能力。

预说明的标识符 (PREDECLARED IDENTIFIERS)

下列是一些预说明的标识符。它们可以由程序员重新定义, 但这种做法被认为有害无益, 应尽量避免。它们是:

• 标准的

ARCTAN	FALSE	OUTPUT	SIN
ABS	FLOAT	PACK	SQR
BOOLEAN	GET	PRED	SQRT
CHAR	INPUT	PUT	SUCC
CHR	INTEGER	READ	TEXT
COS	LN	READLN	TRUE
DISPOSE	MAXIN	REAL	TRUNC
EOF	NEW	RESET	UNPACK
EOLN	ODD	REWRITE	WRITE
EXP	ORD	ROUND	WRITELN

• 扩展的内部特性

ABORT	EVAL	RESULT
BYWORD	HIBYTE	SIZEOF
DECODE	LOBYTE	UPPER
ENCODE	LOWER	

• 字符串内部特性

CONCAT	POSITN	COPYLST	
DELETE	SCANEQ	COPYSTR	
INSERT	SCANNE		
• 系统内部特性			
FILLC	MOVER	FILLSC	MOVESR
MOVEL	RETYPE	MOVESL	
• 扩展 I/O 特性			
ASSIGN	DISCARD	READFN	SEQUENTIAL
CLOSE	FILEMODES	READSET	TERMINAL
• 系统 I/O 特性			
FCBFQQ			
• WORD (字) 类型特性			
MAXWORD	WORD	WRD	
• 高级数组类型特性			
LSTRING	NULL	STRING	

注释 (COMMENTS) 注释是括在 { } 或 (* *) 里的部分, 它可以延续多行。用 “!” 开始的是单行注释, 它可延续到行尾。使用相同定界符的注释不能嵌套。

分隔符 (SEPARATORS) 注释、空格或行结束符都被用作为分隔符, 它们只能出现在两个相邻的数或相邻的保留字和标识符之间, 是不能嵌入到数、保留字或标识符的内部。

在某些情况下, MS PASCAL 可以接受没有分隔符的信息, 且不给出错信息。例如:

100MOD # 50 作为 100 MOD # 50 来接收

100MOD127 看作是 100 跟着标识符 MOD127

分号被用作为语句之间的分隔符, 它不是每个语句的组成部分。

三. 用户标识符

用户标识符, 可以简称为标识符, 是程序设计人员为自己的程序、单元、模块、常量、类型、变量、记录中的域名或记录变体选择域以及语句标号等所选用的一个符号。

标识符的构成规则如下: 标准 PASCAL 语言的规定是, 标识符要由英文字母开头, 后面跟一个或多个英文字母或数字字符构成, 也可以什么都不跟, 仅由一个英文字母构成; 而 MS PASCAL 语言则对它稍加扩展, 规定开头的英文字母之后, 除了可以跟英文字母或数字字符之外, 还可以跟连字符。如 ABC、A12345、A1B2C3、WANG-PC 等等, 都是 MS PASCAL 的合法的标识符, 但最后一个, 即 WANG-PC 在标准 PASCAL 中是非法的, 因为标准 PASCAL 的标识符中不能出现连字符。2A1B、A+C、A.B、A 加 B 等等都是非法的标识符, 它们或者不以英文字母开头, 或在要给出的标识符内出现了不属于英文字母、数字字符和连字符等其它字符。值得特别提一下的是, 标识符内不能出现汉字, 因为微机上的汉字的内部编码不满足标识符的组成规则。除此之外, 我们还特别强调, 不能把 PASCAL 语言的保留字拿来当用户标识符使用, 应避免把 PASCAL 的预说明的标识符用作为用户标识符。

关于语句标号,标准PASCAL语言规定只能用1~9999之间的不带符号的正整数。而MS PASCAL语言则规定,语句标号也可以用标识符表示,并且这些标识符也遵从PASCAL语言对标识符辖域(有效范围)的有关规定。

标识符的长度(字符个数)是任意的,但必须限制在一行之内。MS PASCAL语言规定,一个标识符的前31(标准PASCAL语言规定为8)个字符才是有意义的,长于31个字符的标识符会产生警告性信息。在不致造成重名的情况下,标识符以短些为好,形象化些(指名字和它代表的内容一致或接近)为好,这样更简明、清晰、好记,编译时更节省存储空间。

要传给连接程序(LINK)的标识符也可以有31个字符长。

在反汇编目标代码文件中,那些不被连接程序使用的变量名和过程名会被截成只有六个字符的长度。

运行系统所使用的外部标识符,都是由四个英文字母再跟上“QQ”两个字母组成的。因此,为避免混淆,用户不要用这种格式来定义自己的外部标识符。

四、三种可编译的源文件

MS PASCAL编译程序,可以对PASCAL语言的三种源文件进行编译。这三种源文件分别是程序(PROGRAMS)、模块(MODULES)和单元(UNITS)的实现部分(IMPLEMENTATIONS OF UNITS)。三种源文件都可以含有对单元的接口(INTERFACES)。下面我们分三小节来介绍这三种源文件的特性和各自的用法。

1. 程序(PROGRAM)

PASCAL程序是用PASCAL语言设计出来的、用来实现对给定的数据进行某些运算或操作处理功能的一段源文件。它的特点是本身比较完整独立,每个源程序在经过编译和连接操作之后,都可以在计算机的操作系统控制下独立自主地运行。一般来说,几个不同的程序之间的联系,不会在每个程序之内直接存在。即使在由几个程序“协同”完成一件复杂处理任务的情况下,它们的“协同”问题也是由计算机的使用人员在程序之外来解决的。各程序的编译、连接和运行的独立性是程序的一个重要特点。

程序总是由三个部分组成:

- 程序的首部
- 程序的说明部分
- 程序的执行部分(又叫程序体)

下面是一个简单的小程序。

```
PROGRAM ALPHA (OUTPUT, AFILE, PARAMETER),  
VAR AFILE:TEXT;  
    PARAMETER:STRING (10);  
BEGIN  
    REWRITE (AFILE);  
    WRITELN (AFILE, PARAMETER);  
END.
```

这个程序的第一行,构成程序的首部。ALPHA是这个程序的程序名,它是在比该程序中说明的全部静态标识符更高一层的层次上给出的一个标识符,与程序中预先说明的那

些标识符（如INTEGER、READ、PROCEDURE等等）处在同一层次。所以不能把预先说明的那些标识符用作程序名，否则将造成标识符重名的错误。但程序名却可以在该程序的内部作为另一个标识符重新加以说明和使用，因为这完全满足PASCAL程序的标识符分层说明及使用的有关规定。

一个PASCAL的源程序被编译与连接之后，就可以让它投入运行。此时首先要进行文件与变量的初始化操作，要调用一个ENTGQQ过程，程序名将作为一个公用（PUBLIC）标识符被记忆下来。其它的程序（或模块、单元）就可以在其内部用这个程序名或ENTGQQ过程来调用这个程序的程序体，把它作为一个公用过程（PUBLIC PROCEDURE）来使用。这个方法虽然是可行的，但我们不推荐采用这种办法来运行一个程序，这也是程序的独立性的一种表现。

接在程序名之后在一对括号中给出的是该程序的有关参数。程序参数是解决程序和计算机系统衔接、通信的重要手段。在标准PASCAL语言中规定，程序参数应该是文件类型的变量，是用来建立文件变量和计算机系统中的一个实际文件（通过文件名）的对应关系的一种途径。

程序参数和过程参数是很不一样的。第一，程序参数的说明必须在程序的变量说明部分进行，而不是在程序首部的程序参数表中进行。第二，这些参数所对应的实际内容不会被传送到程序体中。从实质上看，它们是在操作系统中处理的。

如果在程序参数表中没有给出程序中用到文件变量，操作系统就无法自行解决该文件与操作系统管理的实际文件的对应关系。这时，用户可以用PASCAL语言的扩展语句ASSIGN或READFN把一个字符串变量的值作为文件名指定给文件变量。

在程序参数表中，经常给出两个预先说明的文件INPUT和OUTPUT，分别表示终端的键盘输入和终端的屏幕输出设备。对这两个文件参数的处理办法与对别的程序参数处理的办法不一样，主要表现在，

- 它们是作为预先说明的参数处理的，不必在程序的变量说明部分再对它们进行说明。
- 当程序中用到从终端键盘输入、向终端屏幕输出时，INPUT和OUTPUT就必须出现在程序参数表中，否则编译过程中将会给出警告性出错信息。

- 在用正文（文本）文件的过程和函数进行终端入/出操作时，INPUT、OUTPUT这两个文件名可以在READ、WRITE等语句中省略，并且RESET（INPUT）和REWRITE（OUTPUT）两个打开文件的操作是由系统自动完成的。

- INPUT和OUTPUT也可以被重新定义，但在用文本文件的过程和函数执行终端入/出操作时，总是按对INPUT和OUTPUT的在预说明中的初始定义进行处理。

一个程序也可以没有自己的程序参数部分，这时，程序首部就仅由保留字PROGRAM后跟一个程序名组成。在这样的程序中，INPUT和OUTPUT文件是不能使用的。

INPUT和OUTPUT之外的其它程序参数，会在程序初始化期间，通过执行READFN语句取得各自的值。MS PASCAL对标准PASCAL的程序参数部分进行了扩展，也允许把文件类型之外的变量作为程序参数，但规定它们必须属于能用READFN来读它们的值的那些数据类型（如整型，字类型，字符类型，布尔类型，实型、枚举类型和子界类型、指针类型、字符串类型和文件类型），并且必须是一个完整的变量，不能是一个变量的一个组成成分（包括地址类型的.R和.S分量在内）。

在用READFN调用来取得程序的每个参数的值时，是要用到INPUT文件的。在读这

些参数值之前，首先要调用子例行程序PPMFQQ，让READFN去读从名字为PPMUQQ的DOS接口子程序那里返回的字符序列。PPMUQQ是从启动执行PASCAL目标程序的命令行中得到这些字符的。PPMFQQ和PPMUQQ两个子例行程序都要用到PASCAL的目标程序名。需要时，可以让系统给出对程序参数的提示。

以本小节开始给出的程序为例，看一下启动运行它的目标程序的几种方法。假定这一目标程序的文件名为ALPHA·EXE。

第一种方法，给出文件名后直接回车，要全部提示。

```
A)ALPHA✓  
A FILE:DATA. FIL✓  
PARAMETER:ABCDEFGHIJ✓
```

第二种方法，在文件名后面给出参数A FILE的值以后回车，只要最后的一项提示。

```
A)ALPHA DATA·FIL✓  
PARAMETER:ABCDEFGHIJ✓
```

第三种方法，在文件名后给出A FILE和PARAMETER的值以后再回车

```
A)ALPHA DATA. FIL ABCDEFGHIJ✓
```

在用这第三种方法运行文件名为ALPHA·EXE目标程序时，第二个程序参数PARAMETER的值将从直接跟在DATA·FIL之后的空格读起，使给出的字符串的最后一个字符“J”不能被读到。

必须指明，如果给出一个LSTRING类型的程序参数，程序运行时会出现某种含意不清的局面。如果有下面一个程序：

```
PROGRAM PARMS (LN) ;  
VAR LN : LSTRING (10) ;  
BEGIN  
  ;  
END.
```

在对它经过编译和连接之后，用户用下述办法启动执行它的目标程序（文件名为PARAMS·EXE）：

```
A)PARAMS✓
```

此时就没法区分，用户是为LN参数送入了一个长度为零的值呢，还是希望得到一个对LN的提示信息后再送入LN的值。MS PASCAL总按后一种情况进行处理，就是说，在可以给出一个LSTRING类型的程序参数的值的位置上没有给出相应的值时，就在下一行给出对这个参数的输入提示。

当某些应用程序人员期望使用先进的命令行处理功能时，它可以通过调用UNIT U中的PPMUQQ过程来完成。为此，应该把程序首部做成无程序参数的形式。

2. 模块 (MODULE)

MS PASCAL语言中的模块，可以被看成是无执行体的程序。它们的主要用法，是用来把一些独立的模块简便快速地组合成一个更大的程序。每个模块可以被单独编译，但不能单独地执行连接操作，也不能被独立地加以运行。

模块总是由两部分组成：

- 模块的首部

· 模块的说明部分

下面是MS PASCAL的一个简单的小模块。

```
MODULE BETA [PUBLIC];  
VAR X:INTEGER;  
    PROCEDURE GAMMA (VAR Y:INTEGER);  
BEGIN  
    X:=1;    Y:=X*3  
END;  
    FUNCTION DELTA:WORD;  
BEGIN  
    DELTA:=123;  
END;  
END.
```

这个模块的第一行，构成该模块的首部。BETA是这个模块的模块名。它的使用规则和程序名的使用规则完全相同。模块是不能带有自己的参数的。跟在模块名后在方括号中给出的是对这个模块的属性(ATTRIBUTES)说明，上面例子中给出的表明是公用模块。如果没有明确给出属性说明，就认为模块的属性是[PUBLIC]。当不希望指定模块属性时，必须在模块名后给出一对空着的方括号[]。

模块也没有自己的执行体。但作为一个可以独立编译的PASCAL语言的源文件，它又必须以“END.”结束。

模块中的关键部分是它的说明部分。它既可以说明模块内使用的数据类型和变量等等，更重要的是说明组成该模块的若干过程和函数，它们才是将要使用该模块的程序（或其它模块、其它单元）使用的直接对象。

由于模块没有自己的参数和执行体，其它的程序、模块或单元就可以通过该模块的模块名，把其作为无参数的过程加以调用。

在某些情况下，编译程序也会生成对模块进行初始化的必要代码。当模块过程被执行时，将能够执行必要的初始化操作。如果模块内说明了任何文件变量，或者模块内用到了需要进行初始化操作的接口说明，就要求给出进行初始化的必要代码。当编译程序在编译模块的过程中遇到了上述两种情况时，会给出“模块要初始化”的提示信息。除此之外，在模块内说明了模块内要使用的模块变量时（如上面例子中的变量X），模块的初始化也是需要的。这几种情况，应该在使用该模块的程序中，把这模块说明为一个无参数的外部过程。例如：

```
PROCEDURE BETA; EXTERN;
```

这样就能在模块内的任何一个过程被调用之前，使模块过程被唯一的调用一次，以完成模块的初始化操作。如果不属于上述几种情况，模块过程也可以作为外部过程被调用一次，但不会执行模块初始化操作。

模块中的变量，和程序中的变量一样，是不会自动给出属性特性的，只有文件变量的初始化操作是个例外，这在上面已经讲到。

一个程序要用到模块，说到底，是通过调用模块内的过程和引用模块内的函数体现出来的。程序与模块内的过程、函数之间的数据通信也是通过过程、函数的参数实现的。为

此，在程序中必须把用到的有关过程和函数说明为外部过程和外部函数。在分别对程序和模块进行编译之后，再通过连接操作，把模块中的有关内容连接到程序中来，从而得到程序的最终目标程序文件。就这个意义来看，模块很象是为不同程序准备好了的例行子程序库。多个不同的程序可以使用同一个模块，同一个程序也可以同时使用多个不同的模块。模块的应用，解决了把多个不同的模块，简捷地组合成一个更大的程序的实际需要。

下面给出的一个程序，是使用BETA模块的例子。

```
PROGRAM USEMODULE (OUTPUT);
VAR I: INTEGER;
PROCEDURE GAMMA (VAR Y: INTEGER); EXTERN;
FUNCTION DELTA: WORD; EXTERN;
BEGIN
  GAMMA (I);
  I:=I+DELTA;
  WRITELN ('I= '), I; 5)
END.
```

运行结果应为：

I= 128

在这个程序中，把在模块BETA中说明与实现的过程和函数说明为自己的外部过程和外部函数。

在程序体中引用了这个外部函数，调用了这个外部过程。

当分别对它们编译之后，就可以执行连接操作，得到程序的最终运行文件。假如，模块的文件名为M . PAS，程序的文件名为P.PAS，所用磁盘设备名为A，具体操作过程如下：

```
A>PAS1 M, ✓      对模块文件M进行编译
A>PAS2 ✓
A>PAS1 P, ✓      对程序文件P进行编译
A>PAS2 ✓
A>LINK P+M, ✓    对程序和模块编译后的目标文件进行连接
A>P ✓            运行程序，则得到运行结果
```

3. 单元 (UNIT)

MS PASCAL语言中的单元，是用PASCAL语句编写的、其结构非常类似于程序的一段源文件。它可以被单独编译，并可以被别的程序，模块或其它单元使用。单元本身不会独立运行，它只能被调用后才投入运行，就这个意义上讲，它有些象模块，但它是一种比模块功能更强、使用更灵活方便的程序段。

单元通常要由两个部分组成，即单元的接口部分和单元的实现部分。这两部分的用途和内容很不一样。

下面是一个单元的简单例子。

接口部分，文件名为BASEPL。

```
INTERFACE;
```

```
UNIT BASEPLOT (BLACK, WHITE, DRAWLINE);
```

TYPE

RAINBOW= (BLACK, WHITE, RED, BLUE, GREEN);

PROCEDURE DRAWLINE (C: RAINBOW, H, V: INTEGER);

END;

实现部分:

(* \$INCLUDE: 'BASEPL' *)

IMPLEMENTATION OF BASPLOT;

;

END.

接口部分, 是用来表明一个单元的实现部分和使用这个单元的程序、模块、其它单元之间的接口关系的。为此, 可以在接口部分给出常量、类型、高级数组、变量、过程和函数说明。一个单元的接口部分可以被另一个单元的接口、或任何一个程序、模块、单元来使用。

单元的实现部分, 用来给出单元内各过程和函数的执行体。把单元的接口和实现划分成两个独立的部分, 是为了可以把程序和它使用的单元的实现部分分开设计, 单独编译, 使它们不存在制约关系。此外, 这还可以保证用不同语言编写的单元的实现部分, 能与使用该单元的PASCAL语言的程序正确地协同运行。请注意, 只有程序、模块、单元的接口部分和单元的实现部分才能使用一个单元, 单独的一个过程或函数是不能使用单元的。

一个大的程序, 最好组织成一个主程序和若干个单元的形式。这会简化程序设计, 加快程序设计的速度和提高程序设计的质量, 是一种很有效的程序设计手段。

一个正确的 PASCAL 语言的源文件, 应该由零个或多个 (彼此之间用分号隔开) 单元的接口开始, 后面接上一个程序, 或一个模块, 或一个单元的实现部分, 并最终由一个英文句号结束。这三种源文件都是我们本章讨论的 PASCAL 语言的一个可编译对象, 有时我们又称它们中的每一个为一个程序部 (DIVISION)。

在用到接口的程序部中, 接口的内容必须以源码形式出现在程序部的开始部分。一般情况下, 我们总是把每一个接口的内容以源码形式建成单独一个文件, 保存在磁盘上。这样就可以通过MS PASCAL语言的编译命令 \$INCLUDE, 把这个接口文件的内容方便地引进到单元的实现部分与使用这个单元的程序部中。这比起把接口的实际内容直接写到有关程序部中, 更加简便与灵活, 还可以减少书写打字等操作过程中带来的错误。我们在本节开头给出的例子就是这样做的。例中单元的名字叫做BASEPLOT, 它的接口部分的文件名为 BASEPL。在单元的实现部分用 \$INCLUDE: 'BASEPL' 把它的接口文件的内容引入到该实现部分的开始位置。同样, 如果一个程序要使用这个单元BASEPLOT, 它也必须用 \$INCLUDE: 'BASEPL' 把接口文件的内容引入到程序源码本身的开始位置。例如:

(* \$INCLUDE: 'BASEPL' *)

PROGRAM...

USES BASEPLOT...

这就使程序和它用到的单元使用了同一个接口内容。当需要改变接口内容时, 程序和单元实现部分不受影响, 而且不会出现二者用到的接口内容不相一致的错误。相反, 如果把一个接口的实际内容直接写在程序和单元实现部分, 很可能出现书写或打字错误, 或者修改

接口内容时，在一处改过以后忘了改另一处，在二者间就会出现内容不一致的错误。

下面详细介绍单元的接口与实现部分的有关内容。

(一) 接口

接口必须以MS PASCAL的保留字INTERFACE开始，并在需要时，可以在其后面的括号中给出一个版本号。

接下来是UNIT子语句，就是要在保留字UNIT之后给出单元的名字，以表明本接口是属于哪一个单元的。单元的名字是用户选定的一个标识符，它必须符合PASCAL语言对标识符定义的有关规定。单元名字之后，要在括号中给出若干标识符，我们称之为单元的“要素”(CONSTITUENTS)，它们是由该单元提供并使用的，还提供给使用该单元的程序、另外的接口或实现使用的标识符。它们可以是常量标识符、类型标识符、高级数组类型标识符、变量名、过程名和函数名。这些标识符之间不能重名，也不能与源文件中的其它标识符重名。这对使用单元的程序部(DIVISION)是个很大的限制，为此，可以在那里用USES子语句来解决这类矛盾(这在下面就可以看到)。

接下来要通过USES子语句给出该接口要用到的其它单元的名字，和要用到的与那个单元要素相对应的标识符。USES子语句的组成规则是在保留字USES之后给出单元的名字，在名字之后的括号中给出若干个标识符。请注意，USES子语句的格式与UNIT子语句很类似，而且二者之间有密切的关系。前面已经看到，UNIT子语句要在单元名字后的括号中给出该单元提供给程序、模块或其它接口、实现使用的全部标识符，也就是可以“出口”给别的程序部使用的标识符。USES子语句则是指明要使用哪个单元(通过在保留字USES之后的单元名表示)，和怎样接收从所用单元处“进口”来的标识符(用USES子语句中单元名字之后的括号中的全部标识符来表示)。如果二者的括号中的标识符完全相同，也可以在USES子语句中省略这一部分内容，使USES子语句只由保留字USES和单元名字组成。此时，就直接使用在UNIT子语句中给出的全部标识符。UNIT和USES两个子语句中也可以使用不同的标识符，以便解决在使用UNIT的程序部中可能遇到的标识符重名问题。此时，二者间标识符的对应关系，是用标识符在括号中出现的实际位置解决的，这有点类似于过程的形式参数和实际参数的关系。必须指出，USES子语句只能直接在程序、模块、接口或实现的首部之后给出，写在其它位置是不允许的。

在接口的USES子语句后面是接口的重要内容，即对本单元要素的具体说明。

在接口中，对每个单元要素都必须具体说明。说明的格式与程序中的说明部分是类似的，唯一的差别，是对过程和函数的说明只给出它们的首部，真正的说明要到实现部分进行。

还须说明，如果有几个USES子语句，例如“USES A; USES B; USES C;”，可以直接写成“USES A, B, C;”的形式。

在接口部分是不能进行标号说明的。因为GOTO只能在同一个程序部中进行，不同的程序部之间不存在说明与使用同一标号的问题。

最后，是接口的结束问题。每一个接口都要用END跟一个分号来结束。

通常条件下，接口中只说明单元要素，但也允许说明其它的标识符。如果接口用到需要做初始化操作的单元，可以在结束接口的END之前使用关键字BEGIN，以便产生执行这一初始化所必须的代码。

在接口中，还可以对变量、过程和函数给出属性，自动给出的属性是PUBLIC或EXTERNAL，或自动给出EXTERN命令，但要注意，不能使用彼此有冲突的属性。

下面给出一个接口的例子:

```
INTERFACE (3);  
UNIT KEYFILE (FINDKEY, INSKEY, DELKEY, KEYREC);  
    USES KEYPRIM (KFCB, KEYREC);  
    PROCEDURE FINDKEY (CONST NAME:LSTRING;  
        VAR KEY:KEYREC, VAR REC:LSTRING);  
    PROCEDURE INSKEY (CONST REC:LSTRING;  
        VAR KEY:KEYREC);  
    PROCEDURE DELKEY (CONST KEY:KEYREC);  
    PROCEDURE NEWKEY (CONST KEY:KEYREC);  
BEGIN  
END;
```

在这个例子中, KEYFILE 单元要使用 KEYPRIM 单元, 与这二者有关的标识符是 KFCB 和 KEYREC, 是在 KEYFILE 单元的接口中用 USES 子语句给出的。但是对这两个标识符的处理是有差异的, KFCB 是单元 KEYPRIM 的一部分, 但它没出现在这里的 UNIT 子语句中, 因此使用 KEYFILE 单元的程序是不能使用 KFCB 标识符的。过程 NEWKEY 也在接口中给出, 但它也不是单元的一个要素, 所以它既不能出现在任何一个 USES 子语句中, 也不能出现在相应单元的实现部分。这个接口中给出的单元的要素只有四个, 其中三个是 FINDKEY、INSKEY 和 DELKEY 过程, 另外一个是在该接口用到的单元 KEYPRIM 的接口中给出的数据类型标识符 KEYREC。

接口结束之前的关键字 BEGIN 用于产生对 KEYPRIM 单元有关变量进行初始化的代码。如果无此关键字, 则不会产生这些代码。

(二) 实现

单元的实现部分必须以单元自己的接口内容开始。通用的办法是用一个 \$INCLUDE 编译命令来实现。

接下来应该给出在这个实现中将要用到的其它单元的接口内容。往往也用 \$INCLUDE 编译命令实现。

再后面应该是保留字 IMPLEMENTATION OF, 后面跟上单元的名字, 以表明这里给出的是哪一个单元的实现部分。该行内容构成了一个实现的首部。

首部之后是 USES 子语句, 用以给出该实现中用到的其它单元和有关的单元要素。USES 子语句的格式和用法已在讲解接口内容时阐述过了, 这里不再重复。

USES 子语句之后, 是该实现中的标号说明、常量定义、类型定义、变量说明、常量值设置、过程说明和函数说明等。这些说明的前后次序最好按这里给出的次序安排, 但并不苛求。

标号说明和常量值设置(为变量设置初值)只能在实现中进行, 而不能在接口中进行, 而且, 只有在实现的执行体中有 GOTO 语句时才有必要在这里说明标号。这里说明的标号不能用在该实现中说明的过程和函数的内部。

在接口中已经说明过的常量、类型和变量不必在实现中再次说明, 可以在该实现中直接使用它们, 而且在其它有关程序部中也可以被直接使用。在实现中, 只要说明该实现自己用到的常量、类型和变量就可以了。

在接口中作为单元要素给出的全部过程和函数, 是必须在实现中进行定义的。这些过

程和函数的首部中不能给出参数表，也不必给出过程和函数的属性，这些内容将从接口中直接得到。如果不明确给出EXTERN命令，编译程序将自动给出PUBLIC属性。当明确对一个过程或函数给出EXTERN命令时，表明这个过程和函数的实际说明（它的执行体）不是在这个实现中进行的，这就允许用其它语言，例如汇编语言、FORTRAN语言等来写出这个过程或函数的执行体。此时，应该在实现中首先给出用EXTERN命令的过程和函数，以便编译程序对它们进行检查。整个单元也可以都用PASCAL之外的其它语言实现。这时必须在接口中给出正确的调用序列属性，用户还必须懂得过程、函数的参数在调用过程中的内部表示办法。

在实现中，还可以说明只供该实现自己使用的过程和函数，这和一个程序中说明过程和函数是完全一样的。

实现可以有自己的执行体。它是由BEGIN和END之间的若干语句组成的。当应用该单元的程序首次用到该单元时，它的执行体就要被执行，以便完成该单元中所需的初始化操作，包括诸如对文件变量的内部初始化，也包括执行用户的赋初值的各个语句。在初始化时，对编译单元的实现时用到的接口版本号和编译源程序时用到的相应单元的接口版本号，要比较是否相同，以防止运行程序时使用那些已废弃了的单元的实现代码。如果在接口中不给出版版本号，则认为版本号为零。

如果一个程序中用到多个单元，这些单元的实现的执行体，将按程序中USES子语句出现的次序被依次调用。

任何一个实现，都是用在它的执行体之后的一个英文句号结束。实现的执行体可以为“空”，在这种情况下，实现的执行体和结束被写成BEGIN END。

下面我们三个例子来说明程序、单元和模块的使用规则。

第一个例子，是在一个程序中同时用UNIT和MODULE的程序例子。UNIT接口的内容直接写在程序文件的开始位置和UNIT实现部分的开始位置，而不是通过一个\$INCLUDE编译命令把单独的一个接口文件引进到两个源文件中，这是一种不太好的用法。我们在这里这样给出，是为了让读者更好的体会直接写接口的内容和使用\$INCLUDE编译命令的对应关系。

在程序MAIN中，通过给出UNIT UNIT-DEMO的接口部分的内容和通过USES语句给出用到的UNIT的名字UNIT-DEMO，把程序和它用到的UNIT联系起来，在程序的执行体中，通过过程调用语句调用了在UNIT-DEMO中实现的过程UNIT-PROC。

在程序MAIN中，还通过把MODULE MOD-DEMO中说明的过程MOD-PROC说明为程序的外部过程，使程序与MOD-DEMO中的过程联系起来，并在程序的执行部分调用MOD-PROC过程。

下面给出的是程序 MAIN的内容：

```
INTERFACE;  
UNIT UNIT-DEMO (UNIT-PROC);  
  PROCEDURE UNIT-PROC;  
END;  
PROGRAM MAIN (INPUT, OUTPUT),  
  USES UNIT-DEMO,  
  PROCEDURE MOD-PROC, EXTERN;
```

```

BEGIN WRITELN ('Output from Main Program.');
```

MOD-PROC;

UNIT-PROC;

END.

UNIT UNIT-DEMO的实现部分和MODULE MOD-DEMO的内容分别给出如下,

```

INTERFACE,
  UNIT UNIT-DEMO (UNIT-PROC),
  PROCEDURE UNIT-PROC;
END;
IMPLEMENTATION OF UNIT-DEMO,
  PROCEDURE UNIT-PROC,
    BEGIN WRITELN ('Output from UNIT-PROC in UNIT-DEMO.')


END;



END.



MODULE MOD-DEMO,



PROCEDURE MOD-PROC,



BEGIN WRITELN ('Output from MOD-PROC in MOD-DEMO.')



END;



END.


```

下面是程序运行的结果:

```

Output from Main Program.
Output from MOD-PROC in MOD-DEMO.
Output from UNIT-PROC in UNIT-DEMO.
```

第二个例子,是程序用了一个模块,模块又用了一个单元的例子。

在这个例子中,程序LIST用了模块MODCH中说明的过程CHH。这个过程的功能是首先读入整型变量I的值,并且当I的值为0或1时,分别输出两个函数的值。这两个函数则是在单元UNCH中实现的,模块要用这个单元。

下面是程序和模块的源程序清单:

```

PROGRAM LIST (INPUT, OUTPUT),
  procedure chh; extern ;
begin
  chh
end.
(* $include: 'unch' *)
MODULE modch [public],
USES unch (mchrr, urr),
PROCEDURE chh,
var
  il, i: integer,
  ch: char;
```

```

begin
  readln (i);
  while (i = 0) or (i = 1) do
    [ case i of
      0 : [ read (il); writeln (mchrr (il)) ] ,
      1 : [ read (ch); writeln (urr (ch)) ]
    end;
    readln (i)]
  end;
end.

```

单元UNCH的接口和实现部分的内容如下:

```

INTERFACE;
UNIT unch (mchrr, urr);
  FUNCTION mchrr (i: integer): char;
  FUNCTION urr (ch: char): integer;
END;
(* $include: 'unch' *)
IMPLEMENTATION OF UNCH;
FUNCTION urr;
begin
  urr := ord (ch)
end;
FUNCTION mchrr;
begin
  mchrr := chr (i)
end;
end.

```

这个程序的功能,是调用模块MODCH中说明与实现的过程CHH。该过程实现的是让用户用数字0或1选择:是读入一个整型量给I1后引用函数MCHRR得到CHR(I1)的对应字符,还是输入一个字符给CH后引用函数URR得到ORD(CH)的对应编码值。该过程用到的两个函数MCHRR和URR则是在单元UNCH中实现的。为此,程序中要把CHH说明为外部过程。模块中要用\$INCLUDE编译命令和USES子语句建立与UNIT UNCH的联系。

第三个例子是程序PLOTBOX使用单元GRAPHICS,GRAPHICS又使用单元BASEPLOT的例子。程序的功能是在终端屏幕上画一个方框图出来。有关的五个源文件(主程序、两个接口文件和两个实现)的内容给出如下:

```

(* $include: 'GRAPHI' *)
program plotbox (input, output);
uses graphics (MOVE, PLOT);
(* MOVE and PLOT are used identifiers *)

```

```

begin
    MOVE (15,30) ,
    PLOT (15,50) , PLOT (24,50) ,
    PLOT (24,30) , PLOT (15,30)
end.

INTERFACE,
UNIT GRAPHICS (BJUMP, WJUMP) ,
    (* Exported identifiers are BJUMP and WJUMP.
       In the above PROGRAM, MOVE and PLOT are
       aliases for these identifiers. *)
    procedure BJUMP (y, x: integer) ,
    procedure WJUMP (y, x: integer) ;
    (* Procedure headings only above. *)
begin
    (* Begin implies initialization code. *)
end,

(* $include : 'GRAPHI' *)
(* $include : 'BASEPL' *)
(* The following implementation USES
   the UNIT BASEPL. Thus the interface is
   included above and unit used below *)
IMPLEMENTATION of GRAPHICS,
    (* Implementation is invisible to user *)
USES BASEPLOT,
    (* Procedure BJUMP and WJUMP are
       implemented below. Only the identifiers
       are given in heading. The parameter
       lists are given in the interface *)

    procedure BJUMP,
        begin DRAWLINE (BLACK, Y, X) end,
    procedure WJUMP,
        begin DRAWLINE (white, Y, X) end,
begin
    (* Begin initialization *)
    DRAWLINE (black, 0, 0)
end.

```

```

INTERFACE,
UNIT BASEPLOT (BLACK, WHITE, DRAWLINE) ;
  (*   Other identifiers besides procedure
      identifiers can be exported.   *)
  (*   BLACK and WHITE are exported
      constant identifiers.   *)
type RAINBOW= (BLACK, WHITE, RED, BLUE, GREEN) ;
procedure DRAWLINE (c: RAINBOW; y, x: integer) ;
  (*   No begin; therefore, not an initialized unit.   *)
end,

  (*   $include: 'BASEPL' *)
IMPLEMENTATION of BASEPLOT,
type
  direct= (nothing, xup, xdown, yup, ydown) ;
var
  yy0, y0, xx0, x0: integer;
  direction0, direction: direct;
PROCEDURE POSITION (row, column: integer) ;
var ro, co: lstring (2) ;
begin
  if encode (ro, row: 2) and encode (co, column: 2)
    then write (chr (27) , ' [', ro , ', ', co, 'H')
end,
PROCEDURE DRAWLINE,
var i: integer;
begin
  if x<= 0 then x:= 1; if x>79 then x:= 79;
  if y<= 0 then y:= 1; if y>25 then x:= 25;
  if odd (x) then x:= x+ 1;
  case c of
black: [position (y, x) ;
        y0:= y ;      x0:= x;
        yy0:= y;      xx0:= x;
        direction:= nothing] ;
white: [if y=y0
        then if x>x0 then
              [ position (y0, x0) ;
                for i:= 1 to (x-x0) div 2 do
                  write (chr (243) , chr (246)) ;

```



```

        direction:=xup ]
else if x<x0 then
    [ position (y0, x),
      for i:=1 to (x0-x) div 2 do
        write ('—') ;    (*chr (243) , chr (246)*)
      direction:=xdown ] ;
if x=x0
then if y<y0 then
    [ for i:=y to y0 do
      [ position (i, x0) ;
        write (chr (243) , chr (247) ) ] ;
      direction:=yup ]
else if y>y0 then
    [ for i:=y0 to y do
      [ position (i, x0) ;
        write ('|') ] , (*chr (243) , chr (247) *) ;
      direction:=ydown ]
    ]
end,

if direction< >nothing then
    [ position (y0, x0) ;
      if direction0=xup then
        [ if direction=yup then write ('_') else
          if direction=ydown then write ('|') ] else
        if direction0=xdown then
          [ if direction=yup then write ('_') else
            if direction=ydown then write ('|') ] else
        if direction0=yup then
          [ if direction=xup then write ('|') else
            if direction=xdown then write ('_') ] else
        if direction0=ydown
        then if direction=xup then write ('_')
          else if direction=xdown then write ('|') ;
        if (x=xx0) and (y=yy0) then
          [ position (yy0, xx0) ; write ('|') ]
        ] ,
      y0:=y,      x0:=x,
      direction0:=direction
    end
end,

```

这个程序比较长，涉及的内容也比较多，具体实现细节和各个过程的功能要在学完本资料第一部分之后才能完全读懂，我们不在这里进行解释。目前先对与程序和单元的用法有关的问题进行如下说明：

- 程序中通过开始行中的 \$ INCLUDE 编译命令引入了 UNIT GRAPHICS 的接口部分的内容，又在程序首部之后，通过 USES 子语句给出了本程序使用 GRAPHICS 单元的两个过程的过程名 MOVE 和 PLOT。在单元 GRAPHICS 内部，两个过程名分别为 BJUMP 和 WJUMP，而在程序中，是通过 MOVE 和 PLOT 来调用上述两个过程的。这就是我们在前面介绍 USES 子语句时提到的“换名”概念。这样，程序中如果已把 BJUMP 和 WJUMP 标识符作了别的用途，也不影响该程序使用 GRAPHICS 单元，这是避免标识符重名的必要措施。即使不存在重名矛盾，也能为程序设计人员提供选用他比较满意的标识符，为使用 UNIT 中的某些要素有更大的回旋余地。程序中不直接使用 BASEPLOT 单元，也就不必给出与此有关的 \$ INCLUDE 编译命令和 USES 子语句。

- GRAPHICS 单元要用 BASEPLOT 单元，在它的接口部分用 \$ INCLUDE 编译命令引入了 BASEPLOT 的接口内容，在它的实现部分用了 USES BASEPLOT 子语句，但没有给出用到的要素的标识符。这表明，GRAPHICS 在自己的实现部分将通过在 BASEPLOT 接口中给出的要素标识符直接使用它们，即 BLACK, WHITE 和 DRAWLINE, 前两个是枚举类型的常量值，代表颜色（这里已转义使用），最后一个为过程名。

- BASEPLOT 单元的接口中的标识符 RED、BLUE 和 GREEN 没有出现在它的参数表中，只能供单元本身使用，使用该单元的程序、单元等是不能识别与使用它们的。同理，在 BASEPLOT 实现部分说明的类型 DIRECT 和全部变量，以及过程 POSITION 也是该单元自己使用的，是为支持 DRAWLINE 过程而用到的。

- 再次强调，单元中说明与实现的过程、函数，其完整的首部是在接口中给出的，包括过程或函数名，以及它们的参数表。其执行体是在实现部分给出的，在那里，其首部中不能再给出参数表。

- 模块中的过程和函数名，必须在使用它们的程序中被说明为外部过程和外部函数，过程和函数名不能变。单元中的过程和函数名，则是在使用它们的程序中，通过接口内容和程序中的 USES 子语句两处提供出来的，USES 子语句中给出的名字可以和接口中给出的名字不相同，这为程序用不同的名字使用这些过程和函数提供了可能性。

五、编译命令及其用法

元命令

MS PASCAL 向用户提供了许多条编译命令，或称元命令 (METACOMMANDS)。用户可以用这些命令来设置编译程序的任选项和实现条件编译。

可以按它们的作用，把元命令分为如下四类：

(1) 调试和出错管理类 (共15条)：

\$ BRAVE	\$ DEBUG	\$ ENTRY
\$ ERRORS	\$ GOTO	\$ INDEXCK
\$ INITCK	\$ LINE	\$ MATHCK
\$ NILCK	\$ RANGECK	\$ RUNTIME
\$ STACKCK	\$ WARN	\$ TAGCK (WANG IPC)

(2) 列表文件格式控制类 (共10条):

\$ LINESIZE	\$ LIST	\$ OCODE
\$ PAGE	\$ PAGEIF	\$ PAGESIZE
\$ SKIP	\$ SUBTITLE	\$ SYMTAB
\$ TITLE		

(3) 源文件控制类 (共6条):

\$ IF... \$ THEN... \$ ELSE... \$ END		
\$ INCLUDE	\$ INCONST	\$ MESSAGE
\$ POP	\$ PUSH	

(4) 语言层次及优化目标控制类 (WANG IPC) (共9条):

\$ EXTEND	\$ INTEGR: (n)	\$ REAL: (n)
\$ ROM	\$ SIMPLE	\$ SIZE
\$ SPEED	\$ STANDARD	\$ SYSTEM

说明: IBM/PC机和WANG/IPC机上的 MS PASCAL 给出的编译命令不完全相同。WANG/IPC 机上给出的多几条。我们在这些多出的编译命令之后的括号中用标明 WANG IPC 的办法把它们区分出来。它们是 \$ TAGCK 和语言层次和优化目标类中的九条编译命令。

下面先说明使用元命令中的几个问题。

编译命令必须在注释的开始位置给出才有效,连续的多个元命令可以用逗号分开,出现在元命令各组成成分之间的空格、制表符和行结束标记对元命令无影响。因此,{\$ PAGE: 12}和{\$ PAGE: 12}的作用是相同的。出现在注释内容之间的任何元命令不会被识别出来,也不会被处理的。

绝大部分的元命令可以在程序中加以改动,就是说,可以对不同程序段采用不同的控制与管理办法。例如,要重新编译一个大的程序,对原来无错的部分,使用 \$ LIST-,以取消这一部分的清单文件内容,只在需要检查的几段中使用 LIST+,以给出这一部分的清单文件内容。

由于编译程序只保留已处理过的最后一个符号,因此,在处理某些编译命令时,只对它前面的符号起作用。例如:

CONST Q=1; {\$ IF Q \$ THEN...}

由于 \$ IF 不跟在 Q 之后,出现在 \$ IF 元命令中的 Q 就属于未定义的符号了。又如:

X:=P ^; {\$ NILCK+}

这里的 NILCK 只适用于此处的 P ^。

元命令可以引用或设置元变量 (METAVARIABLE) 的值。这些元变量可以是:

- 无类型的: 它们只在被用到时才被引用,例如在 \$ PUSH 中就是这样使用的。
- 整数类型的: 它们是要被设置为某个数值的。例如在 \$ PAGE:10 中就是这样使用的。
- 布尔类型的: 它们是可以有某些数值的。当这个值 > 0 时,表示要启用这个编译选择项,当这个值 ≤ 0 时,表示要取消这个编译选择项。例如,可以用 \$ MATHCK:1 表明要设置、启用算术运算过程中的出错检查,用 \$ MATHCK:0 表示取消算术运算过程中的出错检查。也可用 “+” 和 “-” 分别表明这两种要求,此时要写成 \$ MATHCK+

和 \$MATHCK- 的形式。

• 字符串类型的：它们是要取一个串常量或串常量标识符的值。例如在 \$TITLE: 'May name' 中就是这样用的。

对元变量是按如下办法设置它的值的：

命 令	结 果
\$元变量+	设置其值为 1
\$元变量-	设置其值为 0
\$元变量: 数字	设置其值为数字所表示的整型常量
\$元变量: 标识符	设置其值为常量标识符所代表的值

只有在元命令中的布尔类型和枚举类型常量才被变换为它们的序号值。对布尔类型，FALSE变为 0，而TRUE将变为 1。

在元命令中是不允许使用常量构成的表达式的。然而，用户却可以先说明一个常量标识符，让它等于所期望的常量构成的表达式，然后再在元命令中使用已说明的常量标识符来达到间接使用由常量构成的表达式的效果。

在后面的讲解中，元命令之后跟上+或-表明是启用/取消编译任选项控制，元命令之后跟上:n表示要用一个整型值，元命令之后跟上'TEXT'表示要用字符串，元命令之后什么也不跟表示无类型。

出错情况下的处理，共有五种出错情况。

- (1) 编译期间的警告性错误。
- (2) 编译期间查出的错误。
- (3) 标准PASCAL编译期间和运行期间的错误。
- (4) 程序运行期间可以加以控制的错误。
- (5) 程序运行期间总要查出的错误。

警告性错误是编译程序在用户源程序中查到并已经解决了的错误，它一般不影响程序的正确运行。这类错误包括常用的替换错误和某些程序语法错误。

编译期间查出的错误，是由编译程序或运行期间系统查出的、并向用户提供出错信息的那些错误。在本书附录C中描述了这些错误，这里不再另外分类。例如，“INVALID”（无效的），“ILLEGAL”（非法的），“NOTPERMITTED”（不允许的）等等都属于这类错误。

标准PASCAL语言错误是很少发生并且极难查出的错误，它们是按ISO标准给出的，MS PASCAL不能检查这类错误。

程序运行期间的错误，有些是属于可以按用户意图选择是否要加以检查的错误。但对有些错误来说，即使已规定了不必检查，也很有可能被检查出来。例如，进行代码优化时，非法操作还是可以被检查出来的。另外，一些运行中的错误总是必须检查的，而且不会为此而生成额外的指令代码。只有那些可以控制的错误检查，当启用了要进行检查的有关选择项时才会生成检查这些错误的额外代码。

下面所列运行期间会产生的各项错误总是要检查的：

- 在NEW过程为变量分配堆区空间时出现了堆区溢出。
- 执行DISPOSE过程遇到变量长度不正确。
- 应用接口时，发现版本号不吻合。

- 发现超越函数错误。
- 读入的子界变量的值超出该变量的子界值范围。

下面给出的是一个运行期间可能出现的出错信息的例子:

```
? ERROR: NO ROOM IN HEAP
      ERROR CODE 2001
      PC=1F0C:199, FP=EAFE, SP=EAF4
```

错误编码和出错信息在本书的附录C中给出。PC是程序计数器,FP是堆栈框区指针,SP则是堆栈指针,它们的值是以16进制形式给出的。

必须说明,另一类可能的错误(按理是不应该发生的)是内部错误(INTERNAL ERROR)。出现这种错误表明是编译程序本身出现了问题,此时应找编译程序的厂家或经销商解决。

下面详细阐明每一条编译命令的用途、使用格式,对其中某些命令还给出了简要说明。

这里的阐明,是按前述四类命令分组进行的。在同一组命令中,又用命令名的英文字母排序关系依次讲解。

\$BRAVE

用途:把出错和警告信息显示在屏幕上。如果是\$BRAVE-,则警告信息将不出现在显示屏上,但仍将出现在清单文件中。

格式: \$BRAVE+ (默认为+)
\$BRAVE-

\$DEBUG

用途:启用或取消编译程序所能执行的全部运行的出错检查。

格式: \$DEBUG+ (默认为+)
\$DEBUG-

说明: \$DEBUG检查总是处于启用状态,除非说明为取消状态。

出错检查包括:

```
$ENTRY
$INDEXCK
$INITCK
$MATHCK
$NILCK
$RANGECK
$STACKCK
```

它们也可以单独地予以取消。

\$ENTRY

用途:给调试程序产生过程的入口/出口的调用。

格式: \$ENTRY- (一是默认值)
\$ENTRY+

说明:在过程或函数出错时给出过程或函数的名字,并把它作为运行时错误信息的一部分显示在屏幕上。

如果在\$LINE之后使用\$ENTRY-,

则 \$ENTRY+ 也被取消 (见 \$LINE)。

\$ERRORS

用途: 设置清单每一页上所允许的出错个数。

格式: \$ERRORS: N (默认值为25)。

\$GOTO

用途: 为清单中每一条 GOTO 语句标上 "CONSIDERED HARMFUL" (相当有害) 的警告信息。

格式: \$GOTO- (+是默认值)

\$GOTO+

\$INDEXCK

用途: 检查数组下标值的范围。

格式: \$INDEXCK+ (+是默认值)

\$INDEXCK-

说明: 由于数组下标要频繁的计算和使用, 所以其范围检查从其它子界检查中独立出来了。这种检查对高级数组也要进行。

\$INITCK

用途: 产生代码以设置所有未初始化的整数值为 -32768, 并设置所有的未初始化的指值为 1 (如果启用 \$NILCK的话)。

格式: \$INITCK- (-是默认值)

\$INITCK+

说明: 下列各项不能用 INITCK 来初始化:

- 在 VALUE 段中所涉及到的变量。
- 记录中的域变量。
- 用 NEW 指定的 SUPER ARRAY (高级数组) 的各分量。

\$LINE

用途: 给调试程序生成行号调用, 因此, 运行时系统可以指出出现错误的行号。

格式: \$LINE- (-是默认值)

\$LINE+

说明: 如果用 \$LINE+, 则会自动的生成 \$ENTRY

例子: ? ERROR: NO ROOM IN HEAP

ERROR CODE 2001

LINE 20 IN (PROC NAME) OF (RROC NAME)

PC=1F0C:199, FP=EAFE, SP=EAF4

\$MATHCK

用途: 检查 INTEGER (整数) 和 WORD (字) 值溢出以及零做除数的错误。

格式: \$MATHCK+ (+是默认值)

\$MATHCK-

说明: 检查 INTEGER 的结果时, 不包括对数值正好是 -MAXINT-1 (即 #8000) 的情况。取消 \$MATHCK 并不能保证一定不进行溢出检查, 在第十一章所描述的程序 提供有加减法和乘法函数, 用这些函数实现算术运算时总是允许溢出的。

\$NILCK

用途：检查被引用的指针的出错情况。

格式：\$NILCK+（+是默认值）

\$NILCK-

说明：检查被引用的指针的值是否为：

- NIL（值为0）
- 未初始化（值为1；只能同\$INITCK并用）
- 指针值越界
- 指针指向堆区（HEAP）中的未用的存储区
- 值为奇数（有效的指针是偶数）

当引用指针或把引用指针传递给DISPOSE过程时才进行这些检查，不单独检查地址。

注意：指针引用是指访问由保存在指针中的内存地址所指向的内容。

\$RANGECK

用途：检查子界的有效性。

格式：\$RANGECK+（+是默认值）

\$RANGECK-

说明：这些检查包括：

- 对子界变量的赋值（包括FOR语句的控制变量和结构常量值）
- CHR和BYWORD函数
- 传递给NEW过程的高级数组的上界值
- SUCC和PRED函数
- PACK和UNPACK的下标值
- LSTRING的赋值和数值参数
- 设置类型赋值和数值参数
- 当没有OTHERWISE子句时，检查CASE语句的控制值

\$RUNTIME

用途：供PASCAL运行时使用的特别开关。

格式：\$RUNTIME-（-是默认值）

\$RUNTIME+

说明：当编译过程和函数时，若\$RUNTIME选择启用状态，则“出错位置”将是调用过程和函数的地方，而不是过程和函数本身内部出错的地方。这条命令是与\$LINE和\$ENTRY命令的目标代码相配合的。

\$RUNTIME开关影响栈指针（SP），堆栈框区指针（FP）和程序计数器（PC）。

\$STACKCK

用途：在过程和函数入口处，当压入大于四个字节的参数进入堆栈时，检查堆栈溢出情况。

格式：\$STACKCK+（+是默认值）

\$STACKCK-

\$TAGCK (WANG IPC)

用途：在访问记录的变体域时，检查这些域的值是否合法。

格式：\$TAGCK-（一是默认值）

\$TAGCK+

\$IF...\$THEN...\$ELSE...\$END

用途：实现条件编译。

格式：\$IF 常量 \$THEN...文本1...

\$END

\$IF 常量 \$ELSE...文本2...

\$END

\$IF 常量 \$THEN...文本1...

\$ELSE...文本2...

\$END

说明：如果常量为真（>0），就处理文本1跳过文本2。如果常量值为假（≤0），就跳过文本1而处理文本2。

常量可以是数字常量或是常量标识符，但不能是表达式。

文本是任意的，可以包括行间隔，注释和其它元命令（包括嵌套的\$IF）。

在被跳过的字符串中间如果有元命令则不予处理，除非是有相应的\$ELSE和\$END命令。

例子：(*\$IF CHIP \$THEN *)

CODEGEN (FADC ALL, T1) (*\$END*)

(* \$IF DOSYS \$ELSE*)

IF MYSYS THEN DOITTOIT (*\$END*)

\$INCLUDE

用途：在当前源行之后转到通过文件名指定的源文件上，当该文件结束时，转回。目的是把另外的源文件的内容拿到正在编译的源文件中来一同编译。

格式：\$INCLUDE:“文件名”

说明：任何PASCAL源文件都可以被\$INCLUDE使用。这在使用UNIT时特别有用。文件名是标准的DOS文件名。

\$INCONST

用途：提示用户给出常量说明。

格式：\$INCONST:标识符

说明：如果用户想要编译不同版本的源程序，可以在编译时改变元条件语句中的常量。这样，每次变更时可不必重新编辑源程序。当编译程序遇到\$INCONST:标识符时，它将提示用户输入一个WORD类型的值。

这个效果和处理带有WORD值的CONST段一样。这个WORD值可供表达式和类型子语句使用。

\$INCONST 元命令应该出现在程序段的开头部位，也就是在 LABEL 之后，或在 CONST 的说明之中或之后。

\$MESSAGE

用途：编译期间在屏幕上显示出—段提示信息。

格式: \$ MESSAGE: '文本'

说明: 在进行条件编译时, (\$ IF \$ THEN \$ ELSE \$ END) 要在终端上显示出被编译的是哪段程序内容时, \$ MESSAGE是很有用的。当使用 \$ INCONST来提示用户输入所需要的信息时, \$ MESSAGE也是有用的。

\$ PUSH/ \$ POP

用途: 保存或恢复现行元命令的值。

格式: \$ PUSH

\$ POP

说明: 并不是所有的元命令都可以用 \$ POP来恢复。例如, 如果使用了 \$ LIST的默认值 (即: \$ LIST+不直接地给出在源码中), 在使用 \$ PUSH命令之后接 \$ LIST-, (例如, 用户不想列出用 \$ INCLUDE包含进来的文件的清单内容); 这之后再使用 \$ POP, 则 \$ LIST-仍将起作用。

\$ LINESIZE

用途: 设置程序清单的行长。

格式: \$ LINESIZE: n (79是默认值)

说明: 当要对打印机的行长在80到132个字符范围内进行变动时是有用的。

\$ LIST

用途: 生成源代码清单。

格式: \$ LIST+ (+是默认值)

\$ LIST-

说明: 错误总是要列出的。如果用户想要对一个大的程序作某些修改, 而且还只想列出所修改部分的清单内容, 用户可以通过在程序的修改开始处放上 \$ LIST+, 并在程序的修改末尾处放上 \$ LIST-, 以便只对修改部分产生清单文件。

\$ OCODE

用途: 要或不要反汇编目标代码清单。

格式: \$ OCODE+ (+是默认值)

\$ OCODE-

说明: \$ OCODE命令控制着列出所生成的代码, 其方法和 \$ LIST控制着源代码清单一样。格式基本上象汇编清单, 带有程序代码地址和操作助记符、过程、函数和静态变量标识符在目标代码清单文件中可能被舍去。

由于在PAS 2期间作了代码优化, 所以 \$ OCODE元命令很有可能不会在希望的部位发挥作用。

\$ PAGE

用途: 为下一页设置页号。

格式: \$ PAGE: n

说明: 不会导致跳到下一页。

\$ PAGE

用途: 跳到下一页。

格式: \$ PAGE

说明: 不重新设置行数, 其行数与原 \$ PAGE: n所设置的一样。

\$PAGEIF

用途：如果剩下的行数小于 n 行，就跳到下一页。

格式：\$PAGEIF : n (默认的情况是不起作用)

\$PAGESIZE

用途：设置清单文件中每页的行数。

格式：\$PAGESIZE : n (53是默认值)。

说明：这条命令必须出现在文本的第一行，否则对第一页不起作用。

\$SKIP

用途：跳过几行，或者跳到本页末尾，先遇到哪种情况按哪个办法处理。

格式：\$SKIP : n

\$SUBTITLE

用途：把‘文本’设置为清单文件每页的副标题。

格式：\$SUBTITE: ‘文本’

说明：‘文本’将出现在清单的每一页左上角第二个打印行上。

该命令必须出现在第一行上，以使副标题出现在第一页中。其最大长度应该小于 \$LINESIZE减10，以便能进行适当的调整。

\$SYMTAB

用途：在清单末尾列出程序、过程或函数的变量。

格式：\$SYMYAB+ (+是默认值)

\$SYMYAB-

说明：如果在过程、函数或程序的末尾使用 \$SYMTAB，这个清单将包含有关变量的信息。左边几列给出堆栈框区指针到变量的位移量（指过程和函数），或者确定的内存中的位移量（对主程序和STATIC属性的变量）。

前面有减号的表示的是堆栈框区中的位移量。

该位移量是变量使用的最低地址，这是因为堆栈是从高地址向低地址伸长的。

\$SYMTAB 清单的第一行给出返回地址的位移量（对于主程序来说是0）和堆栈框区的总长度，包括前端暂存区，但不包括代码生成用的暂存区。

对于函数来说，第二行给出返回值的位移地址，长度和类型。剩余各行列出各变量，包括表示它们的类型和属性的关键字。它们是：

关键字	属性
PUBLIC	具有PUBLIC属性
EXTERN	具有EXTERN属性
ORIGIN	具有ORIGIN属性
STATIC	具有STATIC属性
CONST	具有READONLY属性
VALUE	出现在VALUE定义部分
VALUEP	是一个数值参数
VARP	是一个VAR或CONST参数
VARSF	是一个VARS或CONSTS参数
PROCP	是一个过程参数

注意：在用户清单上的符号表中出现的符号，是以类型的字母顺序排列的，而不是按它们在内存中的位置来排列，有关位移量方面的详细内容请参阅数据的内部表示一节。

\$ TITLE

用途：把‘文本’设置为清单页的标题。

格式：\$ TITLE：‘文本’（省略则无标题）

说明：‘文本’将出现在清单的每一页左上角第一个打印行上。

该命令必须出现在第一行上，以使副标题出现在第一页中。

其最大长度应该小于\$ LINESIZE减10，以便能进行适当的调整。

注意：逻辑行可以比一个物理行长，允许把几个元命令放在“第一行”中。

例子：（* \$ TITLE：‘PASCAL PROGRAM’*）

语言层次及优化目标控制类（WANG-IPC）

前面已提到，我们是从四个层次来看待MS PASCAL语言的，即元语言、标准PASCAL语言、扩展PASCAL语言和系统（扩展）PASCAL语言。在我们设计自己的程序时，可以规定使用哪一个层次的PASCAL语言。如果在程序中出现与规定层次不符合的语言特性时，编译程序会给出警告信息。\$ STANDARD、\$ EXTEND和\$ SYSTEM就是用来规定所用语言的层次的。系统中默认的语言层次为\$ SYSTEM+。

\$ STANDARD

用途：设置所用语言层次为标准PASCAL语言，此时，程序中只能使用标准PASCAL语言的特性，不能使用MS PASCAL语言的任何扩展特性。

格式：\$ STANDARD

\$ EXTEND

用途：设置所用语言层次到扩展PASCAL语言。

格式：\$ EXTEND+（+为默认方式）

\$ EXTEND-

\$ SYSTEM

用途：设置所用语言层次到系统PASCAL语言，此时，允许使用MS PASCAL语言全部特性。

格式：\$ SYSTEM+（+为默认方式）

\$ SYSTEM-

\$ INTEGER

用途：设置标准16个二进制位的整数的长度。

格式：\$ INTEGER：2（2是默认长度）

\$ REAL

用途：设置标准实型数的长度。可以设置它为4或为8（个字节）。

格式：\$ REAL：4（4是默认的长度）

\$ REAL：8

说明：可以用这种办法规定整个程序中标准实型数REAL的长度为4或8个字节长。在程序中也能用REAL 4或REAL 8来分别说明不同的实型变量的类型。

\$ ROM

用途：在静态变量初始化时给出警告信息。

格式: \$ ROM+ (+ 为默认方式)

\$ ROM-

MIS-PASCAL语言可以按用户要求进行代码优化,或者不进行代码优化。不进行代码优化可以节省编译时间,加快编译速度,适用于程序的简单调试。为了得到更高质量的目标程序,则应使编译过程中增加代码优化的处理步骤。这种代码优化的目标可以:

- 尽量减少目标程序占用的内存空间,这是默认方式。
- 尽量加快目标程序的执行速度。

这两个目标是彼此矛盾的,每次只能选择其中的一个目标,默认的目标是前者。

\$ SIMPLE

用途:规定对本程序不进行任何代码优化处理。

格式: \$ SIMPLE- (- 为默认方式)

\$ SIMPLE+

\$ SIZE

用途:规定代码优化的目标为尽量减少目标代码占用的内存空间。

格式: \$ SIZE+ (+ 是默认方式)

\$ SIZE-

\$ SPEED

用途:规定代码优化的目标是尽量加快目标代码的执行速度。

格式: \$ SPEED- (- 是默认方式)

\$ SPEED+

下面给出几个使用了各种编译命令的程序实例。

第一个程序例子中,用到了11条编译命令,包括用于调试和出错管理类的、列表文件格式控制类的和语言层次及优化处理控制类的。下面给出了源程序清单、列表文件清单和几种运行结果的内容。

下面是源程序清单,请注意在程序中使用编译命令的方法。

```
(* $TITLE: 'PASCAL PROGRAM' *)
(* $STANDARD *)
(* $LIST- *)
(* $SIZE *)
program tst1 (input, output)
const max=10;
type range= 1..max;
      intval=array [range] of integer;
var i: integer; sum: integer; (* $INTEGER: 2 *) n: 1..30;
    int: intval;
begin
    (* $LIST+ *)
    (* $WARN- *)
    read (n); (* $RANGECK+ *)
    sum:=0;
```

```

    for i:=1 to n do
        begin read (int [i] ); (* $INDEXCK+ *) end
    for i:=1 to n do sum:=sum+int [i] ; (* $MATHCK+ *)
        (* $LIST+ *)
    writeln ('sum=' , sum : 8 ) ;
end.

```

下面给出对源程序进行编译后得到的清单文件的内容，请注意源程序中那些用于对列表文件格式控制的编译命令的作用效果。

清单列表

```

PASCAL PROGRAM                                     Page      1
                                                    01-01-84
                                                    00:10:36
JG  IC  Line#   Microsoft MS-Pascal Compiler, MS-DOS 8086
          Version 3.04, 01/83
00      1      (* $TITLE: 'PASCAL PROGRAM' *)
          2      (* $STANDARD *)
10      5      program tst1 (input, output)
          5      -----Warning 164 Insert; /\
10      12      (* $LIST+ *)
          13      (* $WARN- *)
11      14      read (n) ; (* $RANGECK+ *)
11      15      sum:= 0 ;
11      16      for i:=1 to n do
12      17          begin read (int [i] ); (* $INDEXCK+ *) end
11      18      for i:=1 to n do sum:=sum+int[i]; (* $MATHCK+ *)
Symtab  21      Offset Length Variable
          0      44 Return offset, Frame length
          18      2      I                      :Integer Static
          22      1      N                      :Subrng Static
          24      20     INT                     :Array Static
          20      2      SUM                     :Integer Static

```

接下来给出的是该程序运行的几种结果的内容，请注意用于出错检查的有关编译命令对程序运行产生的效果。

```

E: tst1                                     I  结果溢出
5
12345 32 3453 13443 82542
? Error: Signed Math Overflow
Error Code 1115
PC=F1EA:0002, SS=316C, FP=2F5C, SP=F1E0
E: tst1                                     II 数组下标越界

```

12

1 2 3 4 5 6 7 8 9 10 12 1 12

? Error:Signed Value Out of Range

Error Code 2056

PC=2E3E:0161, SS=3166, FP=F1D8, SP=F1DA

E:ts11

■ 子界越界

35

? Error:Data format error in file USER

Error Code 1115

PC = 029A:F1D0, SS=3166, FP=0000, SP=F1D6

对下面的几个程序例子, 不再进行文字解释, 请读者自行阅读与分析。

例2 $y = \sqrt{\sin(x)}$

设 TAG=1 表示 x 用角度给出

TAG=0 表示 x 用弧度给出

(* \$TITLE:'PROG EX2' *)

(* \$PAGESIZE:50 *)

(* \$SYMTAB- *)

program t2 (input, output);

(* \$INCONST:TAG *)

var x, y:real; (* \$REAL:8 *)

(* \$INCLUDE:'FSN' *)

begin (*MAIN PROGRAM *)

read (X);

(* \$IF TAG \$THEN *)

(* \$MESSAGE:'NOW X IS EXPRESSED IN DEGREE,
CHANGE IT INTO RADIAN' *)

writeln ('x=', x:8:4, 'degree');

x:=x*3.1416/180;

(* \$ELSE *)

(* \$MESSAGE:' X IS EXPRESSED IN RADIAN' *)

writeln ('x=', x:8:4, 'radian');

(* \$END *)

y:=sn (x);

writeln ('y=', y:8:4);

end.

\$PAGESIZE:50 指定清单列表每页50行

\$SYMTAB- 取消清单末尾的变量显示

\$INCONST:TAG 提请常量说明

\$INCLUDE:'FSN' 将文件FSN一同编译

```

$IF TAG $THEN { 条件编译，根据TAG值的不同（即 x 的
$ELSE          \ 表示形式的不同）编译不同的语句
$END

```

\$MESSAGE: ' * * ' 编译时给出提示信息

文件FSN 的内容:

```

function sn (p:real) :real;
begin sn:=sqrt (sin (p)) ; end;

```

File: pas1 t2.pas

Microsoft MS-Pascal Compiler, MS-DOS 8086 Version 3.04, 01/83

Object filename [t2.OBJ] :

Source listing [NUL.LST] :

Object listing [NUL.COD] :

Inconst: TAG=1

NOW X IS EXPRESSED IN DEGREE, CHANGE IT INTO RADIAN

Pass One No Errors Detected.

PROG EX2

Page 1

01-01-84

00:31:18

JG IC Line# Microsoft MS-Pascal Compiler, MS-DOS 8086 Version
3.04, 01/83

00 1 (* \$TITLE: ' PROG EX2' *)

2 (* \$PAGESIZE:50 *)

3 (* \$SYMTAB-- *)

10 4 program t2 (input, output) ;

5 (* \$INCONST:TAG *)

10 6 var x, y:real; (* \$REAL:8 *)

0 (* \$INCLUDE: ' FSN' *)

20 1 function sn (p:real) :real;

=21 2 begin sn:=sqrt (sin (p)) ; { 文件fsn 一同编译

10 3 end;

10 8 begin (*MAIN PROGRAM *)

11 9 read (x) ;

10 (* \$IF TAG \$THEN *)

11 (* \$MESSAGE: ' NOW X IS EXPRESSED IN
DEGREE, CHANGE IT INT

11 O RADIAN' *)

11 12 writeln ('x=', x:8:4, 'degree') ;

11 13 x:=x*3.1416/180;

11 (* \$ELSE *)

15 (* \$MESSAGE: ' X IS EXPRESSED IN


```

                                RADIAN' *)
16      writeln ('x=', x:8:3, 'radian') ;
17      (* $END *)
11      18      y:=sin (x) ;
11      19      writeln ('y=', y:8:4) .
00      20      end.

```

变量显示被取消

```

Errors Warn.  Ir Pass One
0      0

```

```

F:t2
50
x = 30.0000degree
y = 0.5071
E:t2
100
x = 190.0000degree
? Error:SQRT of Negative Argument
Error Code 2104
PC = F43E:CD94, SS = 3293, FP = 2F84, SP = EE52
(* $TITLE:'PROG EX2' *)
(* $PAGESIZE:30 *)
(* $LINE+ *)
program t2 (input, output) ;
(* SINCONST:TAG *)
var x, y:real; (* $REAL *)
(* $INCLUDE:'FSN' *)
begin (* MAIN PROGRAM *)
    read (x) ;
    (* IF TAG $THEN *)
    (* $MESSAGE:'NOW X IS EXPRESSED IN DEGREE,
        CHANGE IT INTO RADIAN' *)
    writeln ('x=', x:8:4, 'degree') ;
    x:=x*3.1416/180;
    (* $ELSE *)
    (* $MESSAGE:'X IS EXPRESSED IN RADIAN' *)
    writeln ('x=', x:8:4, 'radian') ;
    (* $END *)
    y:=sin (x) ;
    writeln ('y=', y:8:4) ;
end.

```

在例2中加入\$LINE+当程序运行出错时，指出的程序名有错。

```
E:
E: T2
199
x=190.0000degree
? Error: Sqrt of Negative Argument
Error Code 2104
Line 2 in SN of
PC=F43E: CD94, SS=32A5, FP=2E8B, SI=EE22
E: (* $TITLE: 'PROG EX3' *)
(* $EXTEND= *)
program t3(input, output);
var n: integer4;
    m: word;
begin read(n);
      m:=hiword(n);
      writeln(m:16);
end.
```

例3

\$EXTEND=禁止语言层次为扩展Pascal语言，程序中用了函数hiword，而仍能正确运行，可知WANG PC不进行语言层次检查。

程序执行结果

```
E: t2 3
65478965
          999
E: t3
48579365334
? Error: Data format error in file USER
Error Code 1126
PC=0978: F398, SS=30F2, FP=F398, SP=F39C.
```

习 题

1. 和你了解的其它计算机语言相比，你认为PASCAL语言有哪些特点？沃斯教授提出与实现PASCAL语言的主要目标是什么？

2. MS PASCAL语言和标准PASCAL语言之间的关系如何？应该怎样理解MS PASCAL语言的四个层次？关键保留字、标准保留字（标识符）和用户标识符在用法上的区别是什么？

3. MS PASCAL与标准PASCAL语言相比，进行了哪些功能扩展？“安全”与“不安全”扩展的区别主要表现在哪些方面？

1. 编译控制命令的用途是什么？按功能分为哪几类？怎样应用这些命令？

5. 三类可编译的MS PASCAL语言的源程序都叫做什么名字？在功能上，它们的区别与联系表现在哪些方面？UNIT比MODULE功能更强，使用更灵活，表现在哪些方面？

第二章 简单数据类型和简单程序设计

MS PASCAL 语言的功能是很强的。我们将把程序设计中涉及到的不同内容分成几章讲解。本章只讲属于简单类型的数据的用法，和设计一些很简单的小程序必然用到的几条语句的使用规则。

众所周知，任何一个程序都由数据结构和算法两大类内容组成。因此有人说，数据结构+算法=程序。

讲到数据结构，就必然涉及到数据、数据元素、数据对象等更基本的概念。

数据 (DATA) 是描述客观事物用到的数、字符、图形、颜色，以及所有能输入到计算机中并被计算机的程序所能处理的符号的总称。它们是计算机程序加工、处理的“原材料”，也是计算机程序加工处理后的“最终产品”。例如：进行数值运算时用到的是整数和实数；编译程序或文字处理程序的处理对象是字符串。在处理语音或进行模式识别的计算机系统中，声音和图形也成为数据。数据的含义是广泛的，而不是单指各种数值。

数据元素是数据的基本单位，即数据这个范畴内的一个个体。有时一个数据元素又由若干个数据项组成，数据项往往是数据的最小单位。

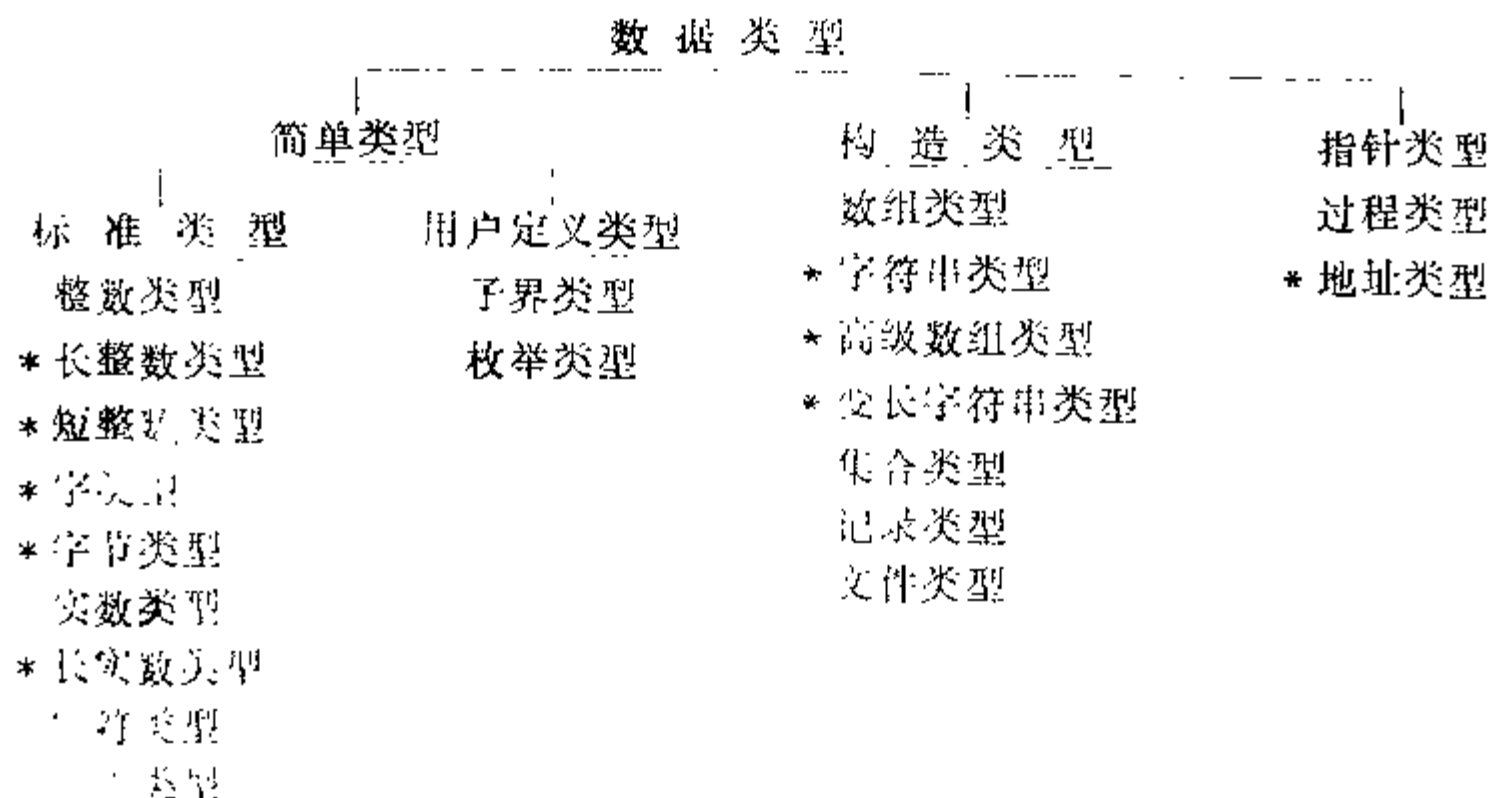
数据对象是指具有相同特性的数据元素的总体，是数据的一个组成部分。例如整数的数据对象是 $\{0, \pm 1, \pm 2, \dots\}$ ；大写字符的数据对象是 $\{A, B, \dots, Z\}$ 。

建立了上述概念后，就可以讨论数据结构的含义了。数据结构，通常是指一批数据元素和表明各数据元素之间的关系结构这样两个方面的内容。

数据类型则是一种程序设计语言所允许使用的变量的种类，换句话说，是用来表示变量可能取的全体值和对它能执行的、能施加的全部运算。

PASCAL 语言的优点之一，是它具有丰富完备的数据类型，就是说，它能清楚、直接地支持程序设计中经常用到的各种数据结构。这对实现结构程序设计是非常必要的。

MS PASCAL 语言的数据类型可列表表示如下：



用 * 标出的数据类型是 MS PASCAL 扩展的类型。

简单类型是最基本的数据类型。它又可以被分成两类：标准类型和用户定义类型。标准类型是在编译程序中预先定义好的，用户可以直接用给出的相应类型标识符来说明变量。用户定义的类型，是要由用户选定类型标识符，并给出它的详细定义。

属于简单类型的数据，可以直接取相应类型的数据值，也可以通过变量名或常量名直接引用。在属于简单类型的数据中，除了实数类型（包括长实数类型）和长整数类型的数据外，都属于有序的数据。实型数据之间、长整数型各数据之间当然也有确定的大小关系，不把它划成有序数据，是指在PASCAL语言中进行的按数据值次序进行的有关操作中，不能使用实数类型的数据和长整数类型的数据。

构造类型，则是由其它数据类型按照一定规则组织而成的数据类型。四种不同的组成规则构成四种不同的构造类型。这些类型，是要由用户选定类型标识符并给出每一个类型的详细定义。

要在PASCAL语句中引用属于构造类型的数据，一般来说，仅用变量名往往是不够的，还必须再给出与构造规则有关的信息。就是说，每次能直接使用的往往不是属于构造类型的一个完整数据，而是它的一个一个的属于简单类型的成分数据。

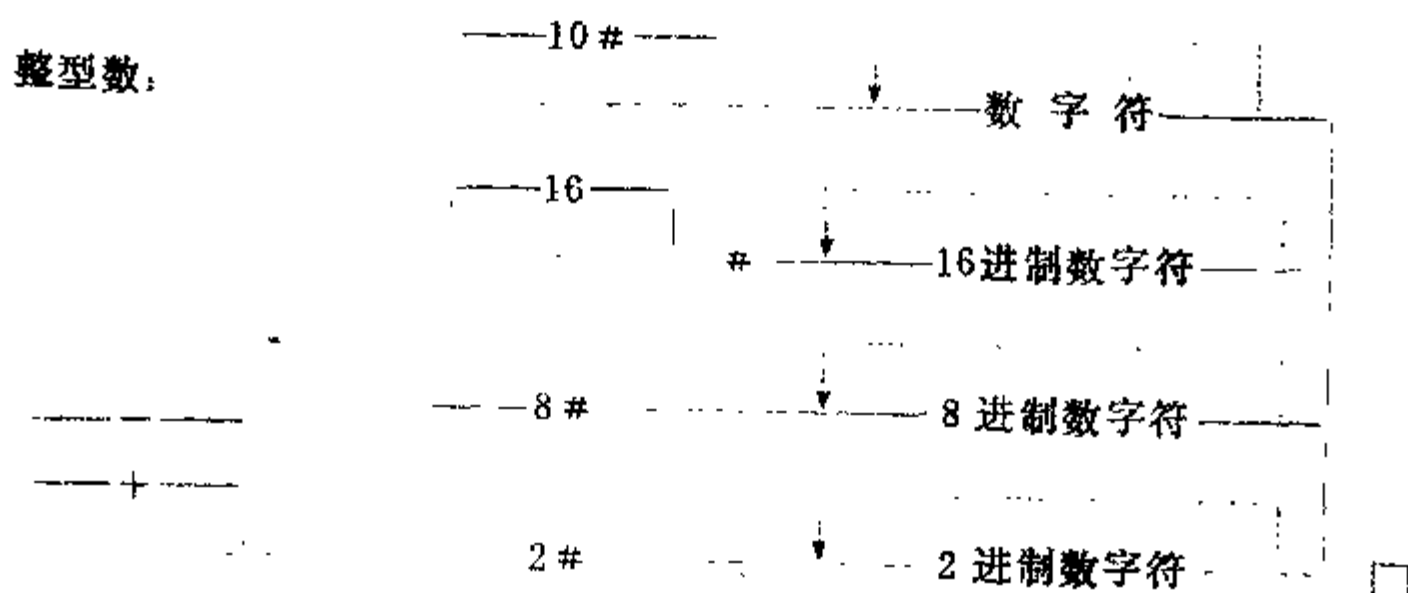
高级数组类型，字符串类型和变长字符串类型是MS PASCAL对标准PASCAL的一种扩展，它们的应用会给程序设计带来许多方便。

指针类型和地址类型也属于简单类型。前者主要用于解决动态数据的建立、赋值、取消等操作，和用于解决特殊结构的数据（如链表、树形数据）的操作。后者主要用于通过数据在内存中的实际地址来使用数据。使用地址类型是一种不安全的办法，但它却是实现某些程序（如计算机的系统程序）和解决某些特殊问题很有效的一种程序设计措施。地址类型在说明与使用等方面，与指针类型是完全不同的，它属于MS PASCAL对标准PASCAL的一种扩展。用了地址类型的程序在不同系列的计算机间是难以移植的。

过程类型只能用来定义过程和函数的形式参数的类型，不能定义程序中的变量。

2.1 整型数据及其运算

整数类型定义的是PASCAL语言的编译程序所支持的正整数，负整数和整数零。它的组成格式如下，



在PASCAL语言里，整数一般以十进制表示，数字符“0”到“9”都可以使用，在MS PASCAL程序中，也可以用二进制、八进制或十六进制表示。

其中正整数和整数零可以不写符号位。一般的整数只能由正负号和数字序列组成，数字序列中间不允许出现任何其它符号。

例如，-13579；0；1375等都是合法的整数。而-157A3；216.0；12，456等都是非法的。因为-157A3不能代表任何整数；216.0则不属于整数，而是一个实数；12，456在财务帐目中使用是允许的，但在PASCAL语言里是非法的。

有一个特殊的整数，用MAXINT表示。它代表某个计算机系统允许的最大整数值。整数的范围是由计算机的字长决定的。因此，不同的计算机系统，MAXINT的数值可以是不相同的。在字长16位的个人计算机中，MAXINT为32767。

整数不仅有最大值，也规定了最小值，个人计算机中最小值为-32767。所以这个系统的整数范围是（-32767到32767）。

必须说明，个人计算机中的最小的整数不是-32768，这与一般的计算机系统的规定是不同的。-32768被用于表示未赋初值的整型量。

如果在程序中给出的整数值，或通过外设输入的整数值超出规定的范围，则是一个错误。前者能在编译过程中发现，后者会在目标程序运行期间发生。

在PASCAL语言中，只能用类型标识符INTEGER来说明整型变量。例如I，J，X都是整型变量，则可以在变量说明部分作如下说明：

```
VAR
```

```
    I, J, X : INTEGER;
```

整型数据之间能进行加、减、乘、和整除、取模等五种算术运算，结果仍为整型数据。整除的操作符为DIV，而通常使用的除法操作符“/”只能用于求实型商。取模的操作符为MOD，求的是整数相除的整余数。

可以通过ORD函数把任何一个属于有序类型的数据值变为一个整型数值。

MS PASCAL语言中还扩展了长整数和短整数另外两种整数类型。长整数由两个内存字（四个字节）组成，即占用32个二进制位，它的取值范围为 $-(2^{31}-1) \sim +(2^{31}-1)$ ，即-2147483647~+2147483647。长整数的类型标识符是INTEGER4。短整数由一个字节组成，即仅占用8个二进制位，它的取值范围为-127~+127。短整数的类型标识符是SINT（或INTEGER1）。要用INTEGER4和SINT（或INTEGER1）来说明长整数和短整数变量。

```
VAR
```

```
    M, N : INTEGER4;
```

```
    P, Q : SINT (或INTEGER1);
```

三种类型的整数之间的关系是明确的。短整数是整数的一个组成部分（用PASCAL语言的术语说，短整数属于整数的子界类型），整数又是长整数的一个组成部分。这些数据的某些性质是相同的，能进行的操作也非常类似。但长整数与整数、短整数的区别是明显的，长整数被认为是无序的数据，因此不能用作数组的下标类型，也不能用作FOR语句的循环控制变量。

MS PASCAL还给出了分别属于整数类型和长整数类型的两个常量名MAXINT和MAXLONG，它们的值分别为32767和2147483647，用户可以在自己的程序中直接引用它们。

MS PASCAL允许用二进制、八进制、十进制或十六进制四种不同的进制方式表示

整型数值。此时要在一个数值的开始位置给出所用的进制基数，这之后用一个字符“#”把进制基数与真正的数值分隔开来。如 $2 \# 10011110$ 、 $8 \# 236$ 、 $10 \# 158$ 和 $16 \# 9e$ 都是合法的整数。若给出一个整数值并且未明确用 # 指定进制，则认为这是用十进制表示的整型数，因此 $10 \# 158$ 和 158 是完全相同的。若只给出 # 而没给进制基数，则认为给出的数据是用十六进制表示的，因此， $16 \# 9e$ 和 $\# 9e$ 是完全相同的。而二进制的数和八进制的数必须给全进制基数、字符“#”和具体数值三部分内容。对整型数据的这些表示方法，也可以用在 MS PASCAL 的 READ 和 WRITE 语句中。上边这一段话，就是对本节开始处给出的描述整型数的语法图所代表的含义的具体说明。怎样看懂和会用 MS PASCAL 的语法图，在本书附录 B 中有简要说明。在学习 PASCAL 程序设计的过程中，有意识的多用语法图，会对自己的学习起到很好的作用。对所学知识的系统性会有更深入的认识。

2.2 字类型、字节类型数据及其运算

支持字类型、字节类型是 MS PASCAL 语言对标准 PASCAL 语言的一个扩充功能。字类型的数据可以取的是个人计算机一个字中 16 个二进制位所表示的数值，即 0 到 65535 ($2^{16}-1$)。使用字类型数据的目的可以概括为：

- 为了使用 32768 到 65535 之间的数值；
- 为了和操作系统或机器结构（汇编语言）进行接口；例如，用 16 个二进制位表示地址，表示汇编语言中使用的 -32768 到 +65535 之间的数据等。
- 为了直接使用一个机器字的 16 个二进制位，以实现诸如计算机中的基本操作（如 AND 操作），这就必须避免使用整型数据（因为可能用到 -32768）。

字类型变量和整型变量是不兼容的两种数据，不能同时出现在一个表达式中，也不能相互赋值。否则将导致一个警告性错误，此时编译程序会随意处理字类型的数据，或把它当作为有符号的数（-32767 到 +32767）或无符号的数（0 到 65535）进行算术运算，但是整型常量可以参加字类型的数据运算。

可以用 WRD 函数把任何属于可数（有字的）类型的一个数据变为字类型的一个值。

在其它 PASCAL 语言中，有时把这儿的字类型叫做无符号整数。

字节类型取的是机器中一个字节的八个二进制位，能表示 0 到 255 之间的任何一个数值。它是字类型数据的一个组成部分。因此它能直接参加字类型数据的运算。

字类型、字节类型的数据之间可以进行加、减、乘、整除、取模五种运算。

用户可以用取序号函数 ORD 把一个字类型的数据转换为整型数据。

在 MS PASCAL 语言中，只能分别用类型标识符 WORD 和 BYTE 来说明字类型和字节类型的变量。例如：

```
VAR  
    W, W1: WORD;  
    B, B2: BYTE;
```

这样，W 和 W1 被说明为字类型的变量。B 和 B2 被说明为字节类型的变量。

与整型数一样，也能用二进制、八进制、十进制或十六进制四种不同的方式给出字类型数据的数值。在用非十进制表示整型、字类型数据时，必须注意下述规定：

MS PASCAL 语言只允许把二进制、八进制或十六进制用于表示整型常量、字型常

量，不允许把这些非十进制的表示法用于表示实型常量。非十进制的表示法也并不意味着是字类型。

非十进制表示方法，是用进制基数跟上一个“#”再跟上相应的符号实现的。例如：

2 # 11011001 8 # 0776 16 # 1AF5 # 1Af5分别表示的是10进制的217、511和6901。当然，在用二进制表示一个整型常量时，跟在#号之后的只能用数字0和1，用八进制时，在#之后只能用0到7这八个数字，用十六进制时，在#号之后可以用0到9这10个数字和大写或小写的A到F（a到f）这6个英文字母。如果在#之前没明确给出进制的基数，MS PASCAL 规定它为16进制。在进位基数有效数字的左边的一个或多个0不起作用。例如，8 # 107 与 008 # 107是一样的。此外，为表示得更清楚，把# AF写成# 0 AF要更好些。

MS PASCAL 还允许对整型，短整型，字类型，字节类型数据进行AND，OR，NOR三种逻辑运算。此时，这些运算是在各对应的二进制位之间进行的，这对于判断一个数据内部表示的每一个二进制位的值，或多个二进制位的值的组合情况是很方便的。例如，把一个整型或字型量和16 # 00FF进行“与”操作，将得到它的低位字节的值，其高位将变成零；若对上两个数进行“或”操作，将得到该量的正确的高位字节的值，低位字节的值将变为全1，即10进制的255。

整型数和字型数的区别，仅表现在对16个二进制位的最高一位的处理不同。对整型数，该位为1表示负号，对字型数，该位为1，代表32768这个数值。

2.3 实型数据及其运算

实数类型的数据包括正实数、负实数和实数零。

一、实数的两种表示法

在 PASCAL 语言中，可以用两种方法表示实数：十进制表示法和科学表示法。

十进制表示法就是小数形式的表示法，人们日常生活中已习惯使用。其格式如下：

实型数：
(小数形式表示)

↓ 数字 . ↓ 数字

其中：

小数点左侧为实数的整数部分；

小数点右侧为实数的小数部分；

正实数和实数零可以不写符号位。

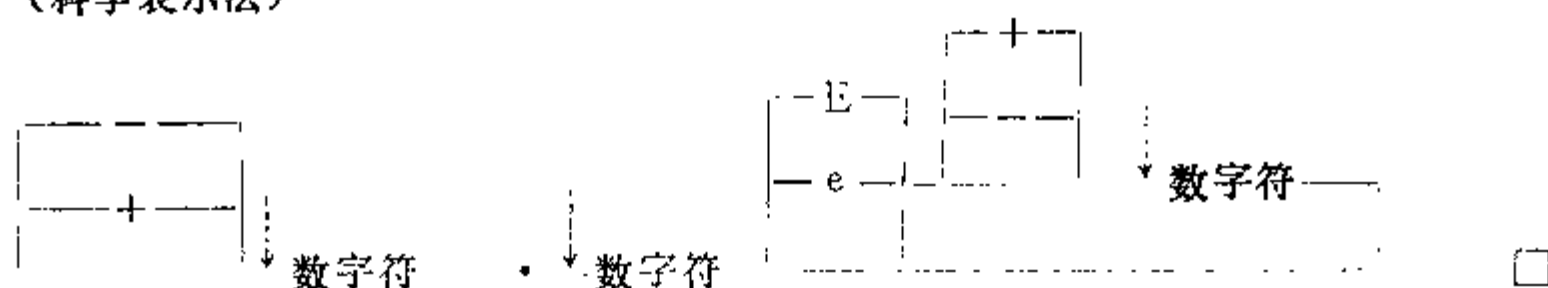
例如：+ 3.5； 0.0； - 0.0； -123456.789等都是合法的实数型数据。

注意，一个实数中只要有小数点，则小数点两侧必须有数字，缺一不可。这跟人们的习惯及其它某些算法语言（如FORTRAN）不同。

例如：-.1567和+123.都是非法的。

科学记数表示法就是指数形式的表示方法，它与计算机的浮点表示是一致的。其格式如下：

实型数：
(科学表示法)



其中字母E表示指数是以10为底，

E的左侧表示实数的尾数部分，包括数符及尾数；

E的右侧表示实数的指数部分，包括阶符及阶码。

正实数及实数零可以不写数符；当一个实数的阶码为正整数或整数零时，可以不写阶符，

实数的尾数必须有，但可以没有小数部分；

实数的阶码必须有，而且必须是整数。

例如，下列实数：

$+5.0E+2$ ， $+5.0E2$ ；

$5.0E+2$ ， $5.0E2$ ；

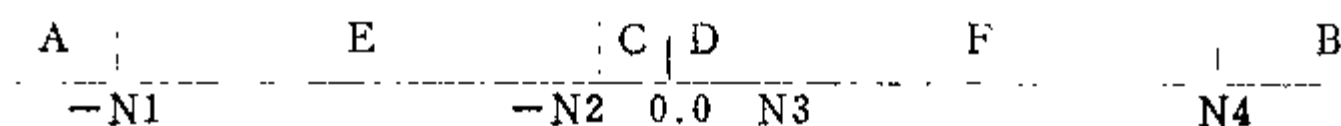
$5E2$ ， 500.0 ；

它们都是合法的，而且是相等的。但最后一个数是用小数形式表示的。

实数 $-6.08E-15$ 和 $-1.0E+2$ 也是合法的，但是，实数 $-6.08E+0.5$ 和 $+E-2$ 则是非法的。前者阶码不是整数，后者缺少尾数部分。

无论用十进制还是用科学表示法来表示实数，它们在计算机里总是用浮点方法来实现的。和整数类似，实数也有一个范围问题。

在数轴上，一个实数可以表示如下：



其中：

$N1, N2, N3, N4$ ，都是正整数；

C区间和D区间实数绝对值太小，接近于零，产生“下溢”，

A区间和B区间实数绝对值太大，超过机器允许范围，产生“上溢”，

E区间和F区间，实数的绝对值适中，是计算机能正确表示的范围。对于不同的计算机系统，表示实数的方法不同，实数的范围也不相同。

必须指出，一个计算机系统的实数范围比整数范围要大得多，这对算术运算很有利。

但是，实数运算存在精度问题，速度比整数运算慢，占用内存也多，所以凡是能用整数运算的，不必用实数运算。

在PASCAL语言中，用类型标识符REAL来说明实型变量。例如

VAR X, Y: REAL;

表示变量X和Y是实数类型变量。

实型数据之间可以进行加、减、乘、除运算,结果仍为实型数据。整型数据是实型数据的一种特例,可以参加实型数据之间的运算。在真正参与运算之前,编译程序会把它首先转换成实型数据的格式。但是实型数据不能直接作为整形数据参加整型数据之间的运算。

MS PASCAL 还扩充了一种长实数类型。它由八个字节组成,即 占用 64 个二进制位。它的类型标识符为 REAL8 (与此对应的 REAL 也可以被写成 REAL4)。它的表示范围和精度都远远超过实整的表示范围和精度。可以用类型标识符 REAL8 来说明长实数类型的变量。例如:

```
RLONG, R2: REAL8;
```

就把 RLONG 和 R2 说明为长实数类型。

正象讨论整数类型时说明的那样,实数类型的数据是长实数类型数据的一个组成部分。它们彼此可以赋值。

REAL4 类型的实型数据的最大值约为 1.7×10^{37} , 有效数字位的精度为 10 进制的 7 位; REAL8 类型的实型数据的最大值约为 10^{46} , 有效数字位的精度达到 10 进制的 16 位。在使用实数时,一定要按实际需要合理选择。

在机器内,实数是用四个字节表示的,其中阶码占八个二进制位,尾数占二十四 个二进制位。长实数是用八个字节表示的,阶码占十一个二进制位,尾数占 53 个二进制位。两种实数的表示方法完全符合 IEEE 国际标准。

2.4 PASCAL 语言中的算术运算

PASCAL 语言有着广泛的运算功能。所谓运算,就是处理由某些运算符和若干个 (也可以是一个)操作数组成一个表达式,产生一定结果的过程。最基本的简单运算格式如下:

操作数 1 运算符 操作数 2

其中,操作数必须属于某种确定的数据类型。一般情况下,操作数 1 和操作数 2 的数据类型应相同,个别运算可以有特殊的要求。运算符决定运算的性质。

MS PASCAL 语言有五种基本运算,列表如下:

	运 算 符	操 作 数 类 型	结 果 类 型
算术运算	+, -, *	整数或实数、字	整数或实数、字
	/	整数或实数	实 数
	DIV, MOD	整数、字	整数、字
	=, <>	标准类型,集合,指针	
关系运算	<, >	标准类型	逻辑型
	<=, >=	标准类型,集合	
	IN	标准类型,集合	
逻辑运算	AND, OR, NOT	逻辑型	逻辑型
集合运算	+, -, *	集 合	集 合
赋值运算	:	除文件类型外 各种数据类型	除文件类型外 各种数据类型

这里先介绍MS PASCAL语言中的算术运算,其它运算将在后续章节中陆续讲解。

算术运算就是对由两个或多个操作数与算术运算符组成的表达式进行的数值计算处理。这里的操作数只能是整型数据、字型数据和实型数据。算术运算符共有六种:加(+)、减(-)、乘(*)、实数除(/)、整数除(DIV)和取模(MOD)。运算的结果或为整型、或为字型、或为实型,由参加运算的数据和算术运算符决定,正如上述基本运算一览表开头部分所表示的。

加、减、乘三种运算符适用于整型、字型和实型三种数据。当参加运算的一对数据属于相同类型时,运算结果的类型与参加运算的数据的类型相同。例如:

$$13+7650=7663 \quad (\text{整型})$$

$$40000-256=39744 \quad (\text{字型, } 40000 \text{ 和 } 39744 \text{ 均属字型, } 255 \text{ 也被作字型处理})$$

$$1.5 \times 2.0 = 3.0 \quad (\text{实型})$$

实数除计算得到的结果一定为实型数据,不管参加运算的两个数同为整型、或同为实型、或一个为整型另一个为实型,也不管计算结果能否被整除得尽,结果都为实型。例如:

$$4/2=2.0 \quad 5.0/2=2.5$$

$$5/3.0=1.666667 \quad 5.0/5.0=1.0$$

必须说明,字类型的数据与整型数据是不同的数据类型,它们不应不经变换就出现在同一表达式中。要让字类型数据和整型数据参加同一运算,可以用函数WORD变一个整型量为字类型的量,也可以用函数ORD变一个字类型的量为整型量。字类型的量参加实数运算更是没有道理的。整型量可以参加实型量的计算,此时加、减、乘的计算结果也一定为实型量。

整除(DIV)和取模,原则上说,只适用于正的整数值,因此,它可以用于正整数和字类型数据的运算,不能用于实型数据,对取值为负的整数进行DIV或MOD运算也是没道理的(编译程序不检查这个错误)。DIV求的是两个正整数值相除的整数商。MOD求的是两个正整数整除后的整余数。DIV和MOD都不允许处于除数位置的数据为零,但允许计算的结果为零。例如:

$$19 \text{ DIV } 4 = 4 \quad 19 \text{ MOD } 4 = 3$$

$$15 \text{ DIV } 20 = 0 \quad 15 \text{ DIV } 5 = 3$$

$$15 \text{ MOD } 20 = 15 \quad 15 \text{ MOD } 5 = 0$$

15.0 DIV 5 或 15 MOD 5.0 等是不允许的,因DIV和MOD不能用于实型数据的运算。

长整数、短整数类型数据参加算术运算的规则与整数类似。字节类型数据参加算术运算的规则与字类型数据类似。这里不再另外说明。

2.5 算术运算程序设计举例

一个计算机程序包括两个实质部分,一个部分用于描述数据,另一个部分用于对给出的数据进行处理。在PASCAL语言中,描述数据由常量定义、类型定义和变量说明来实现,数据处理与操作由一系列的执行语句构成的可执行的程序体完成。

一个PASCAL语言的程序,从总的结构上看,可示意表示如下图所示,

PROGRAM	程序名（程序参数表）， {注释}
LABEL	标号说明
CONST	常量定义
TYPE	类型定义
VAR	变量说明
VALUE	设置变量初值
PROCEDURE	过程说明
FUNCTION	函数说明
BEGIN	语句序列
END.	

我们可以把程序结构框图上给出的内容划分为三大部分。

第一部分是程序的首部，由程序结构框图中第一行组成。它由程序标志 PROGRAM 开头，后跟程序名（是用户给出的一个标识符），最后是在括号中给出的程序参数表。在标准 PASCAL 语言中，在程序参数表中要给出该程序使用的全体内部文件名。MS PASCAL 对此进行了某些功能扩展。

第二部分是程序的说明部分。在标准 PASCAL 语言中，它要由标号说明、类型定义、常量定义、变量说明，过程和函数说明五个部分组成，而且这几部分的次序一般也应遵从这里给出的次序关系。MS PASCAL 语言，还允许在变量说明之后加一个设置变量初值的部分。这比在程序的执行体中用赋值语句设置变量初值能节省程序的运行时间，特别是对那些要经常反复运行，而每次运行又都要设置许多变量的初值的程序来说，这个好处更明显，因为在 VALUE 部分设置变量初值是在编译期间完成的。

第三部分是程序的执行体，是由程序结构框图的最后三行所表示的。它由出现在程序的 BEGIN 与 END 之间的、用分号隔开的一个个语句构成的语句序列组成，用来完成程序的执行功能。END 之后的 “.” 字符用来表明整个程序的结束。

注释则可以出现在程序中任何可以有分隔符的部位。

下面我们将以算术运算为例，看一看非常简单的程序应该怎样设计出来。这一节我们先介绍常量定义和变量说明的有关内容，以及首先遇到的赋值语句、读入语句和输出语句的简单内容。以算术运算为例说明简单程序设计的方法，对那些初次接触程序设计，或者初次接触 PASCAL 语言程序设计的读者来说，也许是最容易学懂、最容易入门的一种选择。

一、常量定义和变量说明

在 PASCAL 语言中，凡是在程序的执行部分用到的常量名和变量，都必须在程序的说明部分加以定义和说明。

（1）常量定义

PASCAL 语言允许用户自己定义一些标识符，用来代表一些常量，其格式如下：
常量说明：

CONST \uparrow 标识符 = 表达式 \square

MS PASCAL又扩充了BREAK、CYCLE和RETURN三个语句。

按语句是否带有标号，可以把语句分为无标号语句和带标号语句。有无标号，对语句本身功能无任何影响，语句标号是为程序运行过程中实现转移用的。

按语句的构成情况，还可以把语句分为基本语句与构造型语句两大类。基本语句是独立的一个简单语句；构造型语句有一定构成规则，它一定含有自己的子语句。标准PASCAL语言总共有十一种语句，考虑到空语句无实际功能，用户可以认为它不存在，所以，标准PASCAL语言实际上有十种功能语句。MS PASCAL又给出了三种扩充语句，从功能方面看，它们都相当于GOTO语句的特定用法。我们在本节将只介绍其中的赋值语句与输入输出语句的部分内容。

赋值语句

赋值语句是最简单的语句。它的功能主要是确定变量的内容。其简单格式如下

——标识符——：——表达式——；

这里的标识符应为变量名。

其中：

变量名必须在变量说明时已经确定；

:=是赋值运算符，它表示将运算符右侧的表达式运算结果送进运算符左侧变量所在的存储单元，就是说把表达式的运算结果赋给变量。

表达式允许为常量、变量、函数、算术表达式或布尔表达式。一般地说，要求表达式的运算结果与变量属于同一数据类型。

例如，在变量说明部分，已经说明X、Y为实型变量，下列赋值语句都是合法的。

Y:=0.5;

X:=Y;

X:=Y+2.5;

X:=SIN(Y+3.5);

Y:=3;

必须指出，PASCAL语言中的赋值语句不同于数学上相等的概念。例如，I和K是整型变量，下述表达式是正确的：

K:=10-K;

I:=I+2;

但是在数学上， $K=10-K$ 是一个方程，其解为5。而 $I=I+2$ 则为一个无解的方程。

此外，赋值语句还常用于为用户定义的函数赋值，即将某一个表达式的值赋给一个函数名，这在函数说明的执行部分是不可缺少的语句。其格式如下：

——函数名——:=——表达式——；

这一用法将要在有关章节另加说明。

在一般情况下，要求变量（或函数）与赋值运算符右侧的表达式具有相同的数据类型。但是有两个特例除外：

（1）变量（或函数）是实数类型，表达式运算结果为整数类型或由整数构成的子界类型；

（2）变量类型是表达式运算结果类型的子界，或表达式运算结果类型是变量类型的子界。（有关子界类型的内容将在以后章节中讲解）。

请注意，在 PASCAL 语言中，把赋值运算符“:=”与等号“=”明显地区别开来，这是一个优点。这样可使用户在进行判断与赋值运算时不会发生混淆。不少初学者，经常在这种地方犯用等号“=”代替赋值操作符“:=”的错误。

2. 输入语句和输出语句

输入和输出语句属于过程调用语句。

过程调用语句是通过给出过程名来调用一个过程使其投入运行的一个语句。如果这个过程带有参数，还要在过程名后的括号中给出调用这个过程时使用的实际参数。过程，就是子程序或例行程序。我们在这里还无法对过程语句做更深入的说明，只能先就 PASCAL 编译程序提供的两个最有用的标准过程 READ 与 WRITE 的用法，作初步讲解。

READ 与 WRITE 是两个标准过程名。这两个过程用于完成在默认的（系统约定的）输入输出设备（在多数微机系统中指的是计算机终端）上的输入输出操作。

与这两个过程相应的，还有 READLN 与 WRITELN 两个标准过程，基本功能与 READ 和 WRITE 相同，但多了个回车换行的处理功能。

实现输入操作的读语句的格式可表示如下：

```
--READ-- ( 变量名 ) --  
--READLN ( 变量名 ) --
```

其中：

READ 和 READLN 是读语句中所用的标准过程名，它们是 PASCAL 语言的标准标识符。

变量名必须在程序说明中预先说明，它的值可以属于整数类型、实数类型。

READ 语句和 READLN 语句的功能基本相同，可以用在程序中需要输入数据值的任何位置，而且可以通过一个读语句一次输入若干个变量的值。两者的区别有以下两点：

(1) READ 语句仅仅要求一个一个数据不断输入，并不要求换行。如果本行还有其它数据，下一个 READ 语句可以接着使用。而 READLN 语句，不仅要求一个一个数据接着输入，一旦本语句读完所要求的数据，则不管本行还剩多少内容，都要跳到下一行去。如果还有下一个读语句的话，它就要从另外一行中读取数据，原来一行中剩余的内容不能再读取了。若一行中输入的数据个数比 READLN 语句中要求的少，则等待继续输入。

(2) READ 语句至少要输入一个数据，而 READLN 语句允许不输入任何数据，只是执行换行的要求。

在一个程序段中，如果有一个以上读语句，要注意有无换行的问题。

举几个例子来说明读语句的应用。假设：A, B, C 和 D 都是整数型变量，已在变量说明中说明过了，执行部分中有下列语句。

```
1. READ (A);
```

终端键盘输入：10✓ (✓代表回车键)

执行结果：A=10

2. READ (A, B, C, D);

(1) 终端键盘输入: 10┐20┐30┐40✓

执行结果: A=10; B=20; C=30; D=40;

(2) 终端键盘输入: 10┐20┐30┐40┐50✓

执行结果同(1), 只是数据50成为多余的。

(3) 终端键盘输入: 10┐20┐30✓

执行结果: A=10; B=20; C=30

因为只输入三个数据, 故第四个变量D没有被赋值, 程序等待用户继续输入。

(4) 终端键盘输入: 10┐20✓

30┐40✓

执行结果: A=10; B=20; C=30; D=40;

3. 下列读语句之间的功能是相同的:

READ (A); READ (B, C, D) 与 READ (A, B, C, D) 功能相同;

READ (A); READLN 与 READLN (A) 功能相同;

READ (A, B); READLN (C, D) 与 READLN (A, B, C, D) 功能相同;

4. 下列读语句之间功能不同:

READLN (A); READLN (B) 与 READLN (A, B) 功能不同;

READLN; READLN (A) 与 READLN (A) 功能不同;

5. READLN (A, B); READLN (C, D);

(1) 终端键盘输入: 10┐20✓

30┐40✓

执行结果: A=10; B=20; C=30; D=40;

(2) 终端键盘输入: 10┐20┐30┐40✓

执行结果: A=10; B=20; C, D等待后面的语句输入。

(3) 终端键盘输入: 10┐20┐30✓

40┐50┐60✓

执行结果: A=10; B=20; C=40; D=50.

30, 60两个数丢失了。

(4) 终端键盘输入: 10✓

20✓

30✓

40✓

执行结果: A=10; B=20; C=30; D=40;

这是因为头一行只输入一个数值, 分配给A, B就只能从第二行上读入20, 同理, C和D要从第三行、第四行上读入30和40。这里用了READLN (A, B) 和READLN (C, D) 不太合理, 即语句表示的意思和实际执行结果不一致。

实现输出功能的写语句的格式可表示如下:

—WRITE— (变 量)

——WRITELN —— (— 项 目 ——) —— □
 ,

其中:

WRITE和WRITELN是写语句中所用的标准过程名,是PASCAL 语言的标准标识符;

项目可以是常量、变量和函数的名称,以及字符串或常量值,此处的项目也可表达式。

如果是常量名,则直接输出该常量的值。

如果是变量名,则会输出该变量的内容。写语句中,变量允许属于任何一种标准数据类型。

如果是字符串,则会在输出设备上重现字符串的内容。这给输出各种表格和提示性信息带来很大的方便(字符串的概念和用法到后面章节介绍)。

如果是函数或表达式,则首先对函数或表达式进行运算,然后输出运算的结果。

在PASCAL语言中,WRITE语句和WRITELN语句的功能基本相同。它们可以用在一个程序中的任何要求输出操作的位置,而且可以在一个写语句中,输出若干个项目。

两种格式的主要区别是:

第一,WRITE语句是一项接着一项地输出,但不换行。WRITELN语句是一项接着一项地输出,输出完最后一项自动换行。

第二,WRITE语句至少必须输出一项内容,而WRITELN语句允许不输出任何内容,仅仅换行。因此,在程序段中如果有以下语句:

WRITELN;

WRITE;

前者WRITELN是合法的语句,后者WRITE则是不允许的。

在程序设计中,数据输出的格式非常重要。在进行数据处理,打印各种表格时尤其要注意这一点。

每一种类型的数据输出时所占的列数称为场宽(field width)。PASCAL语言对各种数据定义了标准场宽。标准场宽受计算机系统的约束,其规定如下:

数据类型	场 宽	
	标准 PASCAL	MS PASCAL
整数类型	10	14
实数类型	22	14
布尔类型	10	14
字符类型	1	1
字符串类型	字符串长度	字符串长度

表中提到的字符串类型，将在后面相应章节解释。

为了得到用户所要求的输出格式，用户可以自己定义场宽，一般有两种方法：

(1) 单场宽。格式如下：

——项目——：——场宽——[]

其中项目允许为常量名，变量名，函数名，字符串或表达式。

例如，已知 I 为整数 125，W 为字类型的 40000，

执行下列语句：

WRITELN (I:6) ; WRITELN (W:6) ;

程序运行后，终端显示器显示如下

□□□125

□□40000 (□代表空格，而不是一个专用字符)

(2) 双场宽。格式如下：

——项目——：——场宽1——：——场宽2——[]

其中项目只能为实数型的变量、函数名或算术表达式，它们运算的结果必须是实数型数据。这种写语句显示的是用小数形式表示的实数型数据。场宽 1 为总列数，它包括符号位，整数部分列数，小数点和小数部分列数。场宽 2 只表示小数部分列数。场宽 1 和场宽 2 都用正整数表示。而且要求场宽 1 大于场宽 2。

例如，已知实数 R 为 -654.321

执行语句 WRITELN (R:10:4) 后，显示器显示出：

□□654.3210

执行语句 WRITELN (R:8:2) 后，显示为：

□□645.32

这时，最后一位的“1”不再显示了。但是必须指出，这个数仍然存在，没有舍去，不影响进一步运算。

请注意，在未定义场宽时，按标准场宽输出，定义场宽后，为保证数据的正确，可以突破不合理的场宽规定。

三、实现算术运算的简单程序设计举例

我们将在已经讲解过的有关知识的基础上，通过几个实际程序的例子，介绍实现算术运算的短小程序的设计方法，并想在可能的条件下，给出程序设计中的某些通用规则。

第一个程序，计算两个已知数的和，并输出计算结果。

PROGRAM SUM (INPUT, OUTPUT) ;

VAR

I, J, SUMJ:INTEGER;

BEGIN

I:=5; J:=10;

SUMJ:=I+J;

WRITELN (SUMJ:5)

END;

□□□□15

这个程序的第一行是程序的首部,是每个程序都应该有的。必须用关键字PROGRAM开头,后面跟一个标识符,用作为程序的标题(程序名),一定要符合定义标识符的有关规定,最好选择得形象化些,反映程序的主要功能,但并不苛求。这里用的程序名为SUM。在程序标题后的括号中给出的是这个程序用到的文件。这里的INPUT代表输入设备,OUTPUT代表输出设备,在大多数的小型、微型机系统中,它们代表终端输入与终端输出部分。

在第二行上的VAR标志开始程序的变量说明部分。PASCAL语言规定,程序中必须对用到的每一个变量都在变量说明部分予以说明,不允许省缺说明,即变量说明是“强制性”的。这里说明了三个整数类型的变量。冒号之前是变量名,冒号之后是这个变量的类型。在这个程序中,三个整型变量是I、J、SUM。

在第四行上的BEGIN和第八行的END之间给出的是程序的执行语句,完成程序的赋值、运算和输出的全部功能。

程序中的所有分号都是语句分隔符。程序的最后的一个点“.”是程序的必要组成部分,是每个程序都必须有的,用来表明程序的结束。

这个程序的第一个执行语句,用于为整型变量I赋值,其值为5。第二个语句用于为整型变量J赋值,其值为10。第三个语句仍然是赋值语句,与第一、二个语句的区别在于,它不再是把一个整型常量值(在我们的例子中为5和10)赋给一个变量,而是要首先计算两个整型变量I与J之和,然后把计算结果赋给另一个整型变量SUM:J。第四个语句,也就是最后一个执行语句是输出语句,完成把计算结果SUMIJ的值输出到终端屏幕上,并完成换行(使屏幕上的光标(CURSOR)移到下一行的行首)。

程序后面用注释形式给出的是这个程序的运行结果,就是I加J运算后的结果为15。请注意,用输出语句输出整型量时指定场宽的办法。

程序中出现的所有的分号“;”都是语句分隔符,起到把前后相邻的两个语句分隔开的作用,它不是任何一个语句本身的组成成分。

第二个程序:计算你随意给出的两个实数的和的10倍是多少,并输出计算结果。

```
PROGRAM MULTY10 (INPUT, OUTPUT);  
CONST M=10;  
VAR  
    R1, R2, R3: REAL;  
BEGIN  
    READLN (R1, R2);  
    R3:= (R1+R2) * M;  
    WRITELN (R3:10:4);  
    WRITELN (R3)  
END.
```

输入: 10.02 100.34

输出: 1110.3600

1.103600E03

这个程序的第一行与前一个程序很类似,只是程序名用的是MULTY10。

这个程序的第二行中的第一个符号是保留字CONST,表明由此开始了程序的常量定义部分。我们这里选用M为常量名(常量标识符),并规定其值为10。常量定义总是在一

个常量名后用一个等号对应给出常量名所代表的值。定义多个常量时，多个常量定义之间用分号隔开。

程序的第三行是变量说明部分。在这里说明了R1、R2和R3都是实型变量。

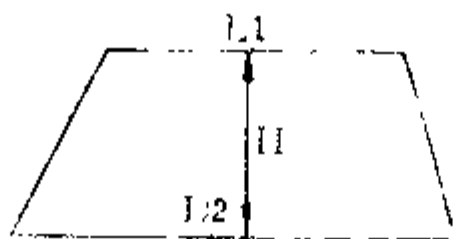
这个程序的第一个语句用于读入R1和R2两个实型变量的值。用读入语句读入变量的值和程序中为变量赋一个常数值是有很大的区别的：读入变量值，是在目标程序运行时才由用户从终端键盘为变量输入一个值；而用赋值的办法时，变量的值是在用户源程序中就确定了的，采用这种方案的程序，在编译连接之后，不管目标程序运行多少次，都会得到同一结果，目标程序运行过程中，无法改变有关的变量的初值。一定要改，则只能修改源程序，如把第一个程序中I的值改为100后，重新执行编译、连接操作之后，目标程序才能计算出100加10的计算结果。第二个程序采用读入变量值的方法，则每当运行这个程序的目标程序时，可以临时输入要参加计算的两个实数的值，对不同的输入值能得到不同的输出结果。程序之后，用注释形式给出了一种可能的输入值和对这组输入计算得出的结果。再一次运行这个目标程序时，可以输入另外一组新的数值，以便得到新的结果。开始学习程序设计时，应该注意到这两种确定变量初值方法的区别和应用场合。

这个程序的第二、第三、第四个语句容易看懂。第二个语句在计算R3的值时，用到了一个常量标识符，其值为整数10，和直接写乘10的计算结果是一样的。请记住，整型量和实型量一起运算时，结果为实型量，这在讨论实数类型时已明确讲过。对第三个语句，请读者注意用输出语句输出实型量时，指定输出格式和场宽的方法。当不指定场宽时，程序将按指数形式输出实型量，如第四个语句所示。

第三个程序用于计算一个梯形的面积：

```
PROGRAM AREA (INPUT, OUTPUT);
VAR
  L1, L2, H, AREA: REAL;
BEGIN
  WRITE ('ENTER THE VALUES OF L1, L2, H: ');
  READ (L1, L2, H);
  AREA := (L1 + L2) * H / 2;
  WRITELN ('AREA = ' AREA : 10 : 4)
END.
(* ENTER THE VALUES OF L1, L2, H, 3.51 4.4 2.0
AREA = 7.9100 *)
```

梯形的短边长为L1，长边长为L2，高为H，从终端读入它们的值后，求出该梯形的面积AREA。



这个程序中需要说明的是WRITE和WRITELN语句中的字符串。字符串，即出现在单引号之内的一串文字。在程序运行时，这些内容被“原样”输出，起提示作用。例如，

第一个执行语句执行后，字符串的内容就出现在终端上，用户知道该输入L1,L2和H的值了。最后一个执行语句能清楚地说明输出的数值是梯形的面积，增加了输出结果的可读性。

这个程序的运行结果可以为：

```
ENTER THE VALUES OF L1, L2, H: 3.51, 4.4, 2.0✓
AREA= 7.9100
```

第四个程序用于计算一个正圆锥体的体积。假定R为底的半径，H为高，则它的体积为：

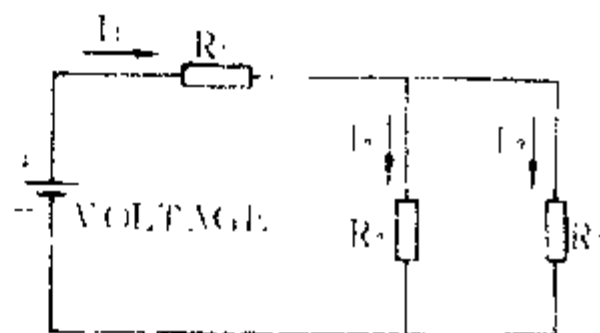
$$VOLUME = \pi R^2 \cdot H \cdot \frac{1}{3}$$

```
PROGRAM VOLUME (INPUT, OUTPUT);
CONST
  PI = 3.14159;
VAR
  R, H, AREA, VOLUME: REAL;
BEGIN
  WRITE ('ENTER THE VALUES OF R AND H: ');
  READLN (R, H);
  AREA := PI * SQR (R);
  VOLUME := AREA * H / 3;
  WRITELN ('VOLUME:', VOLUME:12:5)
END.
(* ENTER THE VALUES OF R AND H: 1.50 5.10
   VOLUME: 12.01658 *)
```

第五个程序用于计算电路中的几个电流值。具体电路如右图。显然，计算电流I1的公式应为：

$$I_1 = VOLTAGE / \left(R_1 + \frac{R_2 \cdot R_3}{R_2 + R_3} \right)$$

计算I2和I3的公式在程序中已表示清楚。



```
PROGRAM CURRENT (INPUT, OUTPUT);
VAR
  VOLTAGE, R1, R2, R3, R, I1, I2, I3: REAL;
BEGIN
  READ (VOLTAGE, R1, R2, R3);
  R := R2 + R3;
  I1 := VOLTAGE / (R1 + R2 * R3 / R);
  I2 := I1 * R3 / R;
  I3 := I1 * R2 / R;
  WRITELN (I1, ' ', I2, ' ', I3)
END.
```

(*5.00 2.00 3.00 4.00

1.346154000E+00 7.692308000E-01 5.769231000E-01 *)

请注意，在PASCAL语言中，除运算符为“/”，并且在两个乘数间的乘运算符“*”一定不能省略不写。

第六个程序用于计算一个物体从500米高空自由落下时所用的时间，和开始落下后2秒、4秒、6秒、8秒和10秒时物体距地面的高度。在这个程序中，取重力加速度为9.80米/秒²，并忽略空气阻力等影响。

```
PROGRAM FALLDOWN (OUTPUT);
CONST
    G=9.80;
VAR
    T, H1, H2, H3, H4, H5:REAL;
BEGIN
    T:=SQRT (500 * 2/G);
    WRITELN ('TOTAL TIME:', T);
    H1:=500-G * SQR (2) /2;
    H2:=500-G * SQR (4) /2;
    H3:=500-G * SQR (6) /2;
    H4:=500-G * SQR (8) /2;
    H5:=500-G * SQR (10) /2;
    WRITELN (H1:10:2, H2:10:2, H3:10:2, H4:10:2, H5:10:2)
END.
(* TOTAL TIME:1.010153000E+01
   480.40    421.60    323.60    186.40    10.00 *)
```

计算落下时间用的公式是：

$$h = \frac{1}{2} g t^2$$

得到，

$$t = \sqrt{\frac{2h}{g}}$$

计算每一给定时刻物体距地面的高度是原来高度减去已落下的距离。

在这个程序和计算正圆锥体体积的程序中，都在CONST部分进行了常量说明。在用常量值的程序中，用合适的常量名替代实际的常量值是一个良好的程序设计习惯，能给程序设计带来很多好处，这一点在前面介绍常量定义时已讲到，请读者从开始编写程序时就注意培养良好的程序设计作风和习惯。

四、MS PASCAL 对 READ 和 WRITE 语句的扩展

MS PASCAL语言，对READ和WRITE语句的功能都进行了扩展。从对数值量的输入和输出处理方面看，这种扩展主要表现在：

1. 在对整型量（也包括短整数和长整数在内）和字类型量（也包括字节类型的量和地址类型的）及 .S 分量）执行输入和输出过程中，也允许使用二进制、八进制和十六

进制表示法。对输入而言，是指允许把从键盘上打入的上述三种非十进制数值传给READ语句中的有关变量，例如16#0B6f、8#377、2#101和#03AB等等。进制基数16、8、2和10要写在字符'#'的左侧，数值跟在#号的右侧。并且规定：

- 10进制可以写成10#...的形式，如10#55。当不明确给出进制基数时，如直接写55，编译程序认为给出的是10进制数值。

- 16进制可以写成16#...的形式，也可以写成#...的形式，如16#0B6f和#0B6f是一样的。即当出现进制标记字符'#'而又没明确给出进制基数时，编译程序把#右侧的数值作为16进制的数处理。为了表示的更清楚，跟在#之后的第一个字符应为数字字符，例如，最好把16#B6f写成16#0B6f。

- 2进制和8进制的数值必须以2#...和8#...的形式给出。

对输出而言，是指把有关的值用二进制、八进制、十进制或十六进制的记数法显示到屏幕上。此时，必须在WRITE语句中，给出每个输出项目所用的进制基数。具体办法是：在原来指明输出项目的场宽的位置，改用:n来指明输出此项目要采用的进制基数。这里的n可以取值为2、8、10或16。当不给出进制基数时，编译程序自动按10进制形式输出。例如WRITE(I)和WRITE(I:n)是一样的。请注意，如果已指明了进制基数，就不能再同时指明场宽。用非十进制表示时，场宽一律默认14，并且有效数字的左端用零字符填充。十进制数有效数字左侧用空格填充。如果用户想取消有效数字左侧的0字符，或想用空格来取代这些0字符，只能在程序中自己处理。

非十进制形式的输入和输出，不能用于实型数据。如果用户在自己的WRITE语句中，为实型数据规定了输出时用的进制数，这是一个错误。但编译程序不检查此项错误，执行真正输出时，也不“理睬”这一规定的进制基数，仍按十进制形式输出。在执行输入时，如果用户用8#123的形式向一个实型量赋值，将把8这个数送给相应的实型变量，#123就不会与该变量有任何关系了，如果用户输入的是#123，系统运行要出错，数据不合法。

2. 在对数值量进行输出时，允许在指明场宽的位置使用正值、负值和数值0来指明场宽。当该值为正时，将按右对齐格式输出；该值为负时，将按左对齐格式输出；该值为0时，将取消此项目的输出中的全部空格。所谓右对齐格式或左对齐格式，是指当指定的场宽比输出的项目的值的有效数字占用的位数更大时，多出的位置要补空格，这些空格补在有效数字的右侧还是补在左侧。空格补在左侧时，称为右对齐输出格式，反之，称为左对齐输出格式。当指定的场宽比实际要用的小时，编译程序会自动按实际要用的场宽输出，以保证正确的输出结果。但对实型量，指定的场宽过小，会影响输出结果的精度。

下面给出的第一个程序和它的运行结果，表明了采用不同的进制执行输入和输出操作时的情形。

```
program jinzhi-test (input, output);
var
  i, j: integer;
  r: real;
  lst, lst1: lstring (16);
begin
  write ('以十进制数输入 I 的值:'); readln (i);
  write ('以十进制数输入 R 的值:'); readln (r);
```

```

writeln (i:10, r:12:4);
writeln;
write ('以八进制数输入I的值:'); readln (i);
writeln (i::2);
writeln (i::8);
writeln (i::10);
writeln (i::16);
writeln;
write ('以八进制数输入R的值:'); readln (r);
writeln (r::8);
writeln;
i:=123;
if encode (lst, i:8) then writeln (lst);
if encode (lst, i:10) then writeln (lst);
if encode (lst, i:16) then writeln (lst);
if encode (lst, i::8) then writeln (lst);
if encode (lst, i::10) then writeln (lst);
if encode (lst, i::16) then writeln (lst);
writeln;
writeln ('取消有效数字前边的零, 输出lst');
for i:=1 to 14 do
    if lst [i] <> '0' then [j:=i; break] ;
move1 (adr lst [j] , adr lst [1] , 15-wrd(j));
lst.len:=lst.len-wrd (j)+1; writeln (lst)
end.

```

以十进制数输入 I 的值:123

以十进制数输入 R 的值:123.456

123 123.4560

以八进制数输入 I 的值: 8 # 123

00000001010011

00000000000123

83

00000000000053

以八进制数输入 R 的值: 8 # 123.456

8.00000000

123

123

123

00000000000173

123

0000000000007 B

取消有效数字前边的零，输出1st

7 B

给出的第二个程序和它的运行结果，用来表明使用不同的场宽值（包括正值、0值和负值）进行数值量的输出时的效用。

```
program rdwrt-twst (input, output);
var
  i : integer ;
  i4: integer4;
  i1: integer1;    r : real ;
  w : word    ;    r14: real4;
  b : byte    ;    r8t: real8;
begin
  i := 20000;    i4 := 1234567890;
  w := 65500; i1 := -120; b := 255;
  writeln('用不同场宽输出整型，字型量时的显示结果');
  writeln('1234567890123456789012345678901234567890');
  writeln('不指定场宽');
  writeln(i, w, i1, b, i4);
  writeln;
  writeln('指定场宽为 6');
  writeln(i:6, w:6, i1:6, b:6, i4:12);
  writeln;
  writeln('指定场宽为 - 6');
  writeln(i:-6, w:-6, i1:-6, b:-6, i4:-12);
  writeln;
  writeln('指定场宽为 0');
  writeln(i:0, w:0, i1:0, b:0, i4:0);
  r := 1.234e12; r4 := 12345.6789e-8;
  r8 := 1234567890.1234567890e200;
  writeln;
  writeln('用不同场宽输出实型，长实型量时的显示结果');
  writeln('12345678901234567890123456789012345678901234567890');
  writeln('不指定场宽');
  writeln(r, r4, r8);
  r8 := 1234567890. 1234567890;
  writeln;
  writeln('指定场宽为 0');
  writeln(r:0);
  writeln(r4:0);
```



```

writeln(r8: 0);
writeln;
writeln('指定场宽为20:10');
writeln(r:20:10);
writeln(r4:20:10);
writeln(r8:24:12);
writeln;
writeln('指定场宽为-20:10');
writeln(r:-20:10);
writeln(r4:-20:10);
writeln(r8:-20:10);
end.

```

用不同场宽输出整型，字型量时的显示结果

1234567890123456789012345678901234567890

不指定场宽

20000 65500 -120 255 1234567890

指定场宽为 6

20000 65500 -120 255 1234567890

指定场宽为 - 6

20000 65500 -120 255 1234567890

指定场宽为 0

2000065500-1202551234567890

用不同场宽输出实型，长实型量时的显示结果

12345678901234567890123456789012345678901234567890

不指定场宽

1.2340000E+12 1.2345680E-04 1.2345681E+209

指定场宽为 0

1.2E+12

1.2E-04

1.2E+009

指定场宽为20:10

1234000000000.0000000000

0.0001234568

1234567890.123500000000

指定场宽为-20:10

1234000000000.000000

0.0001234568

1234567890.1235000000

2.6 字符类型数据及其运算

字符类型数据包括可以打印的全部字符。

在PASCAL语言中，表示字符类型数据的格式如下：

——'——字符——'——□

其中 “'” 称撇，又叫单引号，在字符格式中，必须成对出现，字符必须是可以打印的字符，不能是控制字符。

例如，下列字符是合法的：

'A' '8' '?' ' '

它们分别表示字符A，字符8，字符? 和字符空格。

空格是一个重要的字符，它表示在该位置上留一个空格。为了表示清楚起见，我们在本书中，在表示字符“空格”的地方用“□”代替。实际上打印出来并不存在符号“□”，只是在相应的地方留一空格而已。

要表示字符撇（'）本身，PASCAL语言规定，要连续书写两次（''）。例如，字符''是合法的，它表示字符撇（'）。而'''是非法的字符，它不能表示字符撇（'），也不能表示任何数据。

不同的计算机系统，使用不同的字符集。多数计算机系统用的是128个字符的ASCII码。

ASCII码是美国标准信息交换码（American Standard Code for Information Interchange）的英文字母缩写，共有128个字符，每个字符对应一个字符码。它是国际上用得最普遍的字符集。下面是可打印的ASCII编码表。

可打印的ASCII字符编码表

高位数字	低位数字	0	1	2	3	4	5	6	7
		空 格	!	''	#	\$	%	&	'
4		()	*	+	,	-	.	/
5		0	1	2	3	4	5	6	7
6		8	9	:	;	<	=	>	?
7		@	A	B	C	D	E	F	G
10		H	I	J	K	L	M	N	O
11		P	Q	R	S	T	U	V	W
12		X	Y	Z	[\]	^	_
13		`	a	b	c	d	e	f	g
14		h	i	j	k	l	m	n	o
15		p	q	r	s	t	u	v	w
16		x	y	z	{		}	~	
17									

表中的数字是用八进制形式给出的。

其中字符码十进制的 0 0 至31 (共32个) 以及127所对应的字符是控制字符。控制字符 (如回车、换行等等) 是不能打印出来的。其余95个字符可以打印。因此ASCII码可供打印的字符类型数据是九十五个。MS PASCAL把编码为128~255的128个字符也看成合法字符。

在计算机系统中, 字符是通过对应的字符码来存储的。字符码一般由一个字节(8bits) 组成。一般情况下, 用一个字节存储一个字符。

在PASCAL 语言中, 只能用类型标识符CHAR来说明字符类型变量。例如,

VAR CH1, CH2:CHAR;

表示CH1和CH2是字符类型的变量。必须强调, 字符类型数据只能是一个字符, 不能是一串字符。

例如, 'ABC', '789', 'P8=?' 都不是字符类型数据, 它们不具备字符类型数据的性质。它们是字符串, 属于构造型数据。

字符型数据是非常重要的基础数据之一。因为大部分的慢速输入与输出设备 (如卡片机, 终端, 打印机等) 都是用字符方式工作的。

2.7 布尔型数据及其运算

布尔型数据只有两个值: TRUE和FALSE。它们分别表示逻辑判断的结果是真(TRUE) 或假(FALSE)。这两个布尔型数据也是有次序的, 而且规定:

FALSE<TRUE。

在PASCAL语言中, 要用类型标识符BOOLEAN来说明布尔类型变量。例如

VAR B1, B2:BOOLEAN;

它表示B1和B2是布尔类型的变量。

两个相同类型的数据进行比较, 可以得到布尔型数据。如 $15 > 10$ 为真, ' $B \leq A$ ' 为假。

两个布尔型数据之间可以进行“与”、“或”操作, 其运算规则如下:

“与”操作 真与真, 其结果为真;

真与假, 假与真, 假与假, 其结果均为假。

“或”操作 假或假, 其结果为假;

假或真, 真或假, 真或真, 其结果均为真。

对一个布尔型数据可以“求反”, 或称执行“非”操作, 其规则如下:

“求反”操作 对真求反, 结果为假;

对假求反, 结果为真。

布尔型数据主要服务于逻辑判断和程序中的流程控制。

2.8 枚举类型数据及其运算

枚举类型是一种简单的非标准的数据类型。它通过枚举一系列有序的标识符来定义。这些标识符由用户根据需要自己确定, 它们各自代表一个数据值, 称为枚举类型的元素。它们之间有先后顺序, 可以进行比较。

枚举类型定义的格式如下表示:

——类型名——=——(, 标识符 ,)——□

例如:

```
COLOURS = (RED, YELLOW, GREEN)
```

其中:

类型名和标识符由用户自己确定, 但必须符合 PASCAL 语言对于用户标识符的规定。

用枚举类型说明的变量属于简单类型的数据, 但不是标准类型的数据。要用一个枚举类型来说明变量, 用户就必须首先定义这个类型本身, 而不是由编译程序予以提供的。这也就是把枚举类型划分在用户自定义类型中的理由。

取名枚举类型, 是因为在定义一个具体的枚举类型时, 必须一一列出 (ENUMERATED) 这个类型所包括的全部数据值。而这些数据值本身都是用标识符表示的。

枚举类型是 PASCAL 语言很有特色的一种数据类型, 它采用比较接近人类语言的方式表示有关事务, 以提高程序的可读性, 实现高质量的程序设计。如颜色有红、黄、绿等, 人的性别分为男女; 标准类型应包括整型、实型、字符型、布尔型四种, 一个学校的教师由教授、副教授、讲师和助教四部分人组成。在 PASCAL 程序中, 可以直接用文字表示这些事务, 直观性较强。例如,

TYPE

```
COLOURS = (RED, YELLOW, GREEN),
```

```
SEXTYP = (MALE, FEMALE),
```

```
WEEKDAYS = (SUN, MON, TUE, WED, THU, FRI, SAT),
```

```
IDOBJ = (PROCS, FUNCS, CONSTS, VARS, TYPES),
```

```
STANDARD = (INTS, REALS, CHARS, BOOLS),
```

```
TEACHER = (PROF, ASPROF, LECTURER, ASSISTANT),
```

当然, 上述全部说明都必须在 PASCAL 程序的类型说明部分进行, 即在保留关键字 TYPE 之后进行。在有了上述说明之后, 就可以用它们再去说明有关的变量了,

例如: VAR

```
COLOUR: COLOURS,
```

```
SEX: SEXTYP,
```

```
WEEKDAY: WEEKDAYS
```

```
IDCLASS: IDOBJ,
```

```
STTYPE: STANDARD,
```

```
PERSON: TEACHER,
```

然后, 这些变量就能取相应类型中定义过的值。

我们再把定义枚举类型中的三个特定要求归纳如下,

第一, 每一个具体的枚举类型的名字与它所包括的每一个数据, 都应该是一个标识符, 并且由用户按需要选定 (用户自定义的含意),

第二, 在定义一个具体的类型时, 必须给出它的全部数据值。也就是说, 用户应明确

地列举出这一类型所包括的全部数据值（这就是枚举的含意）；

第三，在一个类型的全部数据之间，应该存在某种确定的关系，如顺序关系，这样才可以在这些数据之间进行比较、寻找等操作（如星期一的下一天是星期二，从一月逐月变到十二月等）。

枚举类型数据只能进行两种运算：赋值运算和关系运算。它不能进行算术运算或逻辑运算。这是枚举类型数据的一个特点。

在赋值运算时，变量名与数据值必须严格地属于同一种枚举类型。

在关系运算时，是根据它们在类型定义中的顺序进行比较，而得到布尔型结果。六种常用的关系运算符都适用于枚举类型。

标准 PASCAL 语言规定，不能用 READ 语句读入枚举类型的数据值，也不能用 WRITE 语句输出枚举变量的值。MS PASCAL 语言中对此进行了某些扩展，也允许用 READ 与 WRITE 语句处理枚举类型的变量，具体用法将在程序实例中讲解。

2.9 子界类型数据及其运算

子界类型是另一个非标准的简单数据类型。它通过两个常量来定义。这些常量由用户根据 PASCAL 语言的语法规则来确定。

子界类型定义的格式如下：

——类型名 —— = ——常量 1 —— ··· ——常量 2 ——[]

其中：

子界名由用户自己确定，但必须符合用户标识符的规定。常量 1 和常量 2 必须是两个属于同一类型的有序的数据。常用的是整数和字符，已经定义为枚举类型的数据也可以使用。布尔类型只有两个值，没必要形成子界类型。实数是无序的数据，不能用来组成子界类型。

常量 1 和常量 2 通常称为下界和上界，下界必须小于上界。例如

TYPE PAN=60..100;

表示 PAN 类型取值范围为 60 到 100。

子界类型数据实际上定义的是整型或字符型或枚举型数据的一个子集，所能执行的操作与原有的数据能执行的操作完全相同，只是取值范围要在规定的上下界之间。其目的是使程序更接近于所处理的事务的本来面貌，多出一种查错的手段与机会。例如，把大学生的年龄说明为 14 到 35 比说明为整型更合情理，把职工的工资说明为 20 元到 850 元比说明为整型好。否则，误把学生的年龄赋为一个负整数，或把职工工资记为百万元，计算机也无法检查出这类错误。

其实 PASCAL 语言中的整型数、字型数、甚至实型数、字符类数据也都是存在的 全部相应类型数据中的一个子界。之所以不能定义实型量的子界，是因为实型量接近于连续的量，存在精度问题，只有属于明确有序的离散量的数据才可能定义它的一个子界类型。

2.10 标准函数及其引用

在标准PASCAL语言中，共定义十七种标准函数，用以实现最通用的数学函数与其它一些特殊的函数运算。这些函数的名称已经确定，作为PASCAL语言中的标准保留字（预说明的标识符），明确建议用户不把它们用作用户定义的标识符。

标准函数，用户不必对它们进行任何说明就可以直接引用。引用格式如下：

——标准函数名——（——自变量——）——□

标准函数都只用一个自变量，其类型与取值范围必须符合相应函数的要求，函数运算的结果类型与具体函数有关。

标准PASCAL语言定义的全部标准函数，列表如下：

标准函数表

标准函数	自变量类型	整 数 (INTEGER)	实 数 (REAL)	字 符 (CHAR)	逻辑型 (BOOLEAN)	文 件 (FILE)
整 数 (INTEGER)		ABS SQR PRED SUCC	TRUNC ROUND	ORD	ORD	
实 数 (REAL)		SIN COS ARCTAN LN EXP SQRT	SIN COS ARCTAN LN EXP SQRT ABS SQR			
字 符 (CHAR)		CHR		PRED SUCC		
逻辑型 (BOOLEAN)		ODD			PRED SUCC	EOF EOLN

下面将对每一个标准函数的功能与使用要求，做一简单说明。我们将用大写的英文字母X表示自变量。

我们把十七种标准函数分为四大类。

第一类：算术运算函数

1. 绝对值函数：ABS (X)

其中X为整数或实数，函数值也为整数或实数。正数的绝对值就是自己，负数的绝对

值为其对立的正数。

2. 平方值函数: SQR (X)

其中X为整数或实数, 函数值也为整数或实数。平方值总是正数, 只有X为零时, 平方值也为零。

3. 平方根值函数: SQR (X)

其中X必须为正整数或正实数, 平方根值一律为实数。对负数不能求平方根值。

4. 正弦函数SIN (X) 和余弦函数COS (X)

其中X为整数或实数, 但函数值一律为实数。X必须用弧度数, 不能直接用角度。

5. 反正切函数: ARCTAN (X)

其中X为整数或实数, 反正切值一律为实数, 也就是弧度数。

6. 指数函数: EXP (X)

其中X为整数或实数, 指数函数值一律为正实数。

它求的是以e为底的指数值。

7. 自然对数函数: LN (X)

其中X为正整数或正实数, 函数值一律为实数。负数不能求其自然对数值。

第二类, 逻辑判断函数

这一类函数是指能够产生布尔类型数据结果的函数。

1. 奇数函数: ODD (X)

其中X必须为整数, X为奇数值, 其函数值为TRUE (真), 否则为FALSE (假)。

2. 行结束函数: EOLN (X)

其中X必须为文件类型变量。在读文件时, 如果读到“本行结束的标记”时, 则函数值为TRUE, 否则为FALSE。

3. 文件结束函数: EOF (X)

其中X必须为文件类型变量。在读文件时, 若读到“文件结束的标记”时, EOF (X) 为TRUE, 否则为FALSE。

关于 EOF (X) 和 EOLN (X) 的详细情况, 将在讲解文件类型的章节中进一步说明。

第三类, 转换函数

对数据类型进行转换的函数, 有四种。

1. 截尾函数: TRUNC (X)

其中X必须是实数类型数据。其函数值是无条件地截去实数的全部小数部分, 取其整数部分。因此截尾函数值总是整数。

2. 舍入函数: ROUND (X)

其中X必须是实数类型数据。其函数值是将其实数的整个小数部分先进行四舍五入处理, 然后取其整数部分。

3. 求序号函数: ORD (X)

其中X一般为字符类型数据, 函数值为该字符所对应的字符码。函数值一定是整数类型数据, 而且范围很小。对个人计算机系统来说, 函数值从32到126。ORD (X) 也用于枚举型数据, 求出该数据在类型定义中的编号。

4. 求字符函数: CHR (X)

其中X必须为一定范围的整数类型数据。函数值为该整数（作为字符码）所对应的字符。

ORD与CHR是多数人不大熟悉的两个函数。它们实现的是变一个字符为它的ASCII码编码值（属于整型量），或变一个整型量（作为一个字符的ASCII码编码值）为一个字符的操作。这在对一串字符进行特定编码与解码，或把一串数字字符变成它对应的二进制数值表示（或相应的逆操作）、或对不能打印的那些控制字符进行输入输出操作，以完成对计算机外部设备的相应控制时，都是必须使用的两个函数。

第四类，进退函数

进退函数是指可以按照一定规律寻找排在该数据前或后的数据的函数，有二种：

1. 前导函数，PRED (X)

其中X允许为整数类型、字符类型、布尔类型或枚举类型数据。函数值为排在X前面的那个数据值。X不能为实数类型数据。

2. 后续函数，SUCC (X)

其中X的规定同前导函数。函数值为排在X后面的那个数据值。

请注意，进退函数要求自变量X属于有序的数据类型。有序的数据必然有界。因此，对“排头”找不到前导值，对“排尾”找不到后续值。

此外，可以发现，前导函数和后续函数互为逆函数，因此：

$$\text{PRED}(\text{SUCC}(X)) = X$$

$$\text{SUCC}(\text{PRED}(X)) = X$$

请读者注意，标准函数的自变量是个非常重要的概念。它不仅可以是变量，而且也可以是常量、表达式，甚至允许是个函数。例如，下列各种函数 $\text{SIN}(X)$ ， $\text{SIN}(15)$ ， $\text{SIN}(X+15)$ ， $\text{SIN}(\text{SIN}(X+15))$ 都是合法的。

2.11 关系运算和逻辑运算

在本章的第四节给出了PASCAL语言中的五种基本运算，并且已在第四节讲过了其中的算术运算，在第五节讲过了赋值运算。在这一节，我们将再介绍两种基本运算，即关系运算和逻辑运算。

一、关系运算

关系运算就是对两个表达式（操作数）和关系运算符组成的布尔表达式进行的比较与判断处理。

操作数可以是任何一种标准数据类型，也可以是表达式，但必须是同一类型，否则，无法进行比较和判断处理。关系运算的结果一定是布尔型数据。

常用的关系运算符有六种：

$=$ ， $<>$ ， $<$ ， $>$ ， $<=$ ， $>=$ ，分别代表相等，不等，小于，大于，小于等于、大于等于。

此外，集合类型数据有五种关系运算符：

$=$ ， $<>$ ， $>=$ ， $<=$ 和 IN ，将在以后讲解。

二、逻辑运算

逻辑运算就是对若干个（也可以是一个）操作数通过逻辑运算符组成的布尔表达式进

行的逻辑处理。其中操作数必须是布尔型数据，运算的结果也一定是布尔型。MS PASCAL语言还允许对整型、字型数据进行逐位的逻辑运算。

标准PASCAL语言中，逻辑运算符有三个：NOT，AND，OR，分别表示“非”、“与”和“或”的意思。MS PASCAL语言中又增加了“异或”运算，操作符为XOR。

AND、OR和XOR要求两个操作数，而NOT只要一个操作数。

表达式的组成规则：操作数允许是常量、变量、因子和项，甚至是一个复杂的表达式。其中所说“因子”，可以是常量、变量和函数等。而所谓项，可以由因子通过 $*$ ， $/$ ， DIV ， MOD 和AND等组成。简单表达式由项通过 $+$ ， $-$ ，OR等组成。复杂表达式可以由简单表达式通过关系运算符构成。

在进行基本运算时，要注意以下三点：

(1) 数学上的表达式 $A \geq B \geq C \geq D$ ，在PASCAL语言里必须写成 $(A \geq B) \text{ AND } (B \geq C) \text{ AND } (C \geq D)$ ；

(2) 逻辑运算的操作数与逻辑运算符之间，必须留一空格。例如： $X \text{ AND } Y$ 不能写成 $X \text{ ANDY}$ 。

如果操作数本身是一个布尔表达式，则必须用圆括号将其括起来。

(3) 执行基本运算的优先级如下：

- 圆括号 $(\dots (\dots (\dots) \dots) \dots)$ ，按照由内到外逐层展开的规则进行；
- 逻辑非运算NOT；
- 乘除类运算：AND， $*$ ， $/$ ， DIV ， MOD ；
- 加减类运算：OR，XOR， $+$ ， $-$ ；
- 关系运算： $=$ ， $<>$ ， $<$ ， $>$ ， \leq ， \geq ，IN；
- 在同一级，按照自左至右的顺序依次执行。

2.12 过程类型

过程类型与其它所有类型都不一样。它既不能用来在程序的TYPE说明部分说明过程类型，也不能用来在程序的VAR说明部分说明过程类型的变量。IBM PASCAL语言中不存在过程变量。它只能用来在过程的首部中说明过程的过程形式参数或函数形式参数。而过程形式参数和函数形式参数，是PASCAL程序中的过程、函数可以使用的四种形式参数中的两种。

说明过程形式参数、函数形式参数的办法是，在相应位置给出过程、函数的首部。包括过程、函数名，它们本身可能有的参数说明，而形式函数参数，还必须给出函数的结果类型。由此可以看出，说明过程、函数形式参数与说明过程、函数时给出过程、函数的首部是类似的。不同之处在于，此处的参数名由用户选定，不必与将来用到的实际参数名相同，而且过程、函数形式参数本身的参数的标识符是不起作用的，编译程序不处理它，真正用到的只是这些标识符的类型说明。

例如，

```
PROCEDURE ZEROPOINT (FUNCTION FUN (X:REAL):REAL);
```

这就在过程ZEROPOINT中说明了一个函数形式参数，参数名为FUN，其结果类型为REAL。函数形式参数FUN本身的形式参数为X，此处的标识符X是不被处理的，只有

它的类型说明REAL是有用的。

有关过程类型的更详细的内容，我们将在介绍过程和函数的一章中进行讲解。

2.13 简单程序设计举例

在这一节，我们将介绍使用已学过的全部简单类型数据进行简单程序设计知识，也包括使用标准函数等方面的内容。

第一个程序是使用四种标准类型数据和有关的几个标准函数的例子。

```
PROGRAM USEFUNCTION (INPUT, OUTPUT);
VAR
  I:INTEGER;
  R:REAL;
  CH:CHAR;
  B:BOOLEAN;
BEGIN
  READ (I);
  WRITELN (PRED (I), I, SUCC (I));
  WRITELN (ODD (I), ' ', ODD (PRED (I)));
  WRITELN (SQR (I), ' ', SIN (I:6:4));
  CH:=CHR (I);
  WRITELN (CH, ORD (CH));
  WRITELN (PRED (CH), CH, SUCC (CH));
  B:=ORD (CHR (I))=1;
  WRITELN ('ORD (CHR (I)) = I:', B);
  I:=ABS (I);
  R:=SQRT (I);
  WRITELN (R:10:4, TRUNC (R), ROUND (R));
END.
```

(* 60

59 60 61

FALSE TRUE

3300 --0.3018

60;

, <=

ORD (CHR (I)) = I:TRUE

7.7460 7 8 *)

这个程序的第一个语句用于读入一个整型变量的值，并把它传给变量I。或简单地说，读入整型变量I的值。接下来的三个语句是输出I为自变量时有关几个函数的运算结果。例如，输入的I值为60，则第一个语句输出的I的前导值，I的值和I的后续值应为59、60和61。第二个输出语句输出I是否为奇数，I的前导值是否为奇数的结果应该是FALSE

和TRUE。由此可以看到,是可以WRITE语句输出布尔型变量的值的。我们提醒读者,在标准PASCAL中,却不能用READ语句输入布尔型变量的值。第三个输出语句输出I的平方值和正弦值。

这个程序的第二小部分,基本上是围绕字符类型变量进行处理的。那里的第一个语句,是把整型变量I的值60作为字符的ASCII码编码值找出对应的字符,并把它赋给字符变量CH,此时CH为'<'。接下来的两个输出语句输出这个字符的值和字符的编码值(此时为<和60),以及字符<的前导字符、<字符和<的后续字符(此时为'=')。再接下来的语句把判ORD(CHR(I))是否等于I的结果赋给布尔变量B,此时B的值为TRUE。再下面一个语句输出一个字符串和B的值,其格式为ORD(CHR(I))=I, TRUE。

这个程序的最后一小段是求I的绝对值并把它赋给I,这样不管原来I的值为正为负,操作后的I值一定不为负(只可能是零或正值)。接下来求I的平方根值。并把它赋给实型变量R。最后一个输出语句用来输出R的值,对R值按舍掉小数、按四舍五入办法处理小数部分得到的两个整数值。

在程序后面给出的是程序的运行结果。程序中每个输出语句都是用WRITELN实现,每个输出语句输出的内容各占一行,很容易看明白。

第二个程序是应用枚举类型的一个例子。

```
PROGRAM ENUMERATED (OUTPUT),
TYPE
    WEEKDAY= (SUN, MON, TUE, WED, THU, FRI, SAT);
VAR
    I:INTEGER,
    TODAY, YESTERDAY, TOMORROW:WEEKDAY;
BEGIN
    TODAY:=TUE;
    YESTERDAY:=PRED (TODAY);
    TOMORROW:=SUCC (TODAY);
    I:=ORD (TODAY);
    WRITELN (ORD (YESTERDAY), I, ORD (TOMORROW),
    TOMORROW>TODAY)
END.
(*      1      2      3      TRUE *)
```

这个程序中不用输入数据,所以程序的首部中不必给出INPUT文件。

这个程序定义了一个枚举类型WEEKDAY,即一个星期中的七天是由星期日到星期六组成的。

在变量说明部分,说明了TODAY, YESTERDAY 和 TOMORROW 三个WEEKDAY类型的变量。

在程序的执行部分,用了一个赋值语句把TUE赋值给TODAY,即今天是星期二。下面的两个赋值语句是说昨天是今天的前一天,明天是今天的后一天。再下面的一个赋值语句求出TODAY的取值在WEEKDAY类型数据中的排列位置。在标准PASCAL语言中,

枚举型变量的值既不能用READ语句读入，也不能用WRITE语句输出。为了看枚举变量的取值情况，得绕过小弯子。

有三个标准函数能用于枚举型数据，它们是 PRED、SUCC 和 ORD。这个程序中都用到了。ORD 用于求一个枚举值在全体枚举值中的排列位置。头一个枚举值的序号为 0，以后顺序加 1。如 TUE 在 WEEKDAY 中的序号为 2。因此，我们可以用输出枚举值序号的办法了解枚举变量的现行值。对枚举型变量可以进行赋值与关系运算。在进行了上述说明之后，读者很容易看出该程序的运行结果为：

```
1      2      3      TRUE
```

MS PASCAL 对枚举类型变量的输入和输出进行了扩展，也允许用 READ 语句读入枚举变量的值，只是输入时，仅能用字类型的一个数值作为某一枚举值的序号的办法读入枚举变量的一个值，目前尚不支持直接以字符串形式输入枚举变量的元素的标识符。当用 WRITE 语句输出一个枚举变量的现行值时，也是把该变量的序号值作为字类型的量进行输出。其实，MS PASCAL 对布尔变量也扩充了用 READ 语句读入其值的功能，处理办法与处理读入枚举变量值的办法一样。标准 PASCAL 和 MS PASCAL 都允许用 WRITE 语句输出布尔变量的值，而且输出的是布尔变量的两个枚举元素标识符 FALSE 和 TRUE（常量标识符）。

下面是一个用 READ 和 WRITE 语句处理枚举型变量、布尔型变量入/出问题的程序例子。注意这个程序中的 WHILE B DO [...] 这个语句是目前尚未讲到的，先照样子使用即可。

```
Program eb (input, output);
var
  e: (ab, cd, ef, gh, ij, kl); b: boolean;
begin
  b:=true;
  while b do
    [write ('Enter the enumerated variable value:');
      readln (e);      writeln (e, ord (e) );
      case e of
        ab: writeln ('ab');
        cd: writeln ('cd');
        ef: writeln ('ef');
        gh: writeln ('gh');
        ij: writeln ('ij');
        kl: writeln ('kl');
      end;
      write ('Enter then boolean variable value:');
      readln (b); writeln (b, ord (b) );
    ]
end.
(*
```

Enter the enumerated variable value: 4

4 4

ij

Enter then boolean variable value: 1

TRUE 1

Enter the enumerated variable value: 2

2 2

ci

Enter then boolean variable value: 0

FALSE 0

*)

习 题

1. 说明下列数据的类型, 如果有错误, 请指明数据表示中的错误所在, 如果是表达式和函数, 请在可能情况下指出计算结果的值。

2547, -1.58, TRUE, -5763425, E+15, -1.3E-0.5,

1.31357, 3765, 423, MAXINT, FALSE, SIN(0.5), PRED

(TRUE), SQRT(-25), CHR(ORD(65)) 'A', '123', '?', SQR

(-5), '','', ' ', 125>124, 125 MOD 13, 125 DIV 5, 125/5

PRED(1.5), SUCC(PRED(18)), 1.25E-50, TRUNC(1.5/

0.15), TRUNC(20/13), ROUND(20/13)

1. 2E200, 65000, 6500000, WRD(123), ORD(123)

2. 说明使用枚举类型数据与子界类型数据的好处, 并试用这两种数据类型表示下述数据(直接用中文表示)。

家庭成员由祖父、祖母、父亲、母亲、儿子、女儿组成, 具有家长身份的当然是前四人。

教室中的设备包括: 桌子、椅子、黑板、板擦四种, 其中价钱在8元以上的设备是前三种。

电视机的价钱在300元至2400元之内变化。

正常成年人的体重在45公斤至120公斤之间。

我国的工作日为星期一到星期六, 星期日是休息的一天。

3. 分析下述表达式哪些是正确的, 哪些是错误的, 如果可能请求出运算结果。

$1 * 2 + 4 / 4 - 12 \text{ MOD } 6$

$\text{NOT } (12 = 12) \text{ OR } (12 = 12)$

$12 = 12 \text{ AND } 12 < > 12$

$\text{PRED}(12 = 12) = \text{NOT TRUE}$

$(12 = 12) > = (12 < > 12)$

$16 - 16 \text{ DIV } 4 * 4 + \text{SQR}(5)$

$\text{CHR}(65) = 'A'$

$\text{SUCC}(4 * 'A') + 1 = 'B'$

4. 检查下面一个程序中的错误（有16处）：

```
PROGRAM ERROR16 (INPUT)
CONST A = 5;
      B, C = 2.7;
TYPE D: (A13, B24, C35, D46) ,
VAR
      ERRNO = INTEGER;
      A, E, F: REAL;
      DD: D;
      CH: CHAR
      I: I:100
BEGIN DD := A13, CH = 'A';
      I := 125;
      B := 2.7 ERRNO := 15;
      WRITELN (I, B, 15, 'ABC') ;
      READ (I, DD) ;
      CH := 'ABC';
END.
```

5. 编写一个程序，读入一名同学六门课程的考试成绩，输出其平均分数。

6. 设计一个程序，同时使用INTEGER1、INTEGER、INTEGER4和BYTEWORD五种类型的数据，测出它们的最大值、最小值，输出时的默认场宽，并检查它们之间的关系，即同时出现在同一个表达式中，是否需要变换类型，会不会产生溢出等。

7. 设计一个程序，同时使用REAL2和REAL4两种数据类型，测出它们的最大值，最小值，输出时的默认场宽和数据的精度。检查它们出现在同一个表达式中的情形。

第三章 程序设计中的流程控制

PASCAL是进行结构化程序设计比较理想的一种语言。

结构化程序设计，是在六十年代末、七十年代初发展起来的一种更科学的程序设计方法。它的出发点是：

(1) 必须以一种易于理解和能维护的方式设计和书写程序；

(2) 通过把要解决的问题细分为容易处理的若干元素，用相对独立的程序模块实现相应的功能，从而得到一个可靠的软件。

结构化程序设计的优点是明显的。首先，它能用有限种类的控制结构去表示程序的控制逻辑，每个控制结构仅执行一种操作，实际上，使用顺序、分支和循环这些结构以及它们的组合和嵌套等，就能实现任意复杂的算法。第二，模块与层次的结构（过程和函数），能非常好地支持自顶向下设计方法中的功能逐步细化，和程序设计过程中的分开来平行设计，最终合起来调试运行工作方式。

在本章中，我们将首先讲解结构化程序设计中用到的三种控制结构，即PASCAL语言程序设计过程中，经常用到三种基本程序结构：顺序结构、分支结构和循环结构。这里说的三种程序结构，是从程序中的一组语句（程序段）执行的控制逻辑和操作功能来划分的。顺序结构，是说这段程序中的语句按书写的次序依次执行；分支结构，是指按给定的条件从两支或多支程序段中选取其中一支加以执行的情况；循环结构，是指按一定条件或按给定的次数重复执行一段程序的情况。这三种程序结构是进行任何程序设计都会用到的。可以说，用这三种结构以及它们的组合和嵌套等，一定能实现任意复杂的算法。

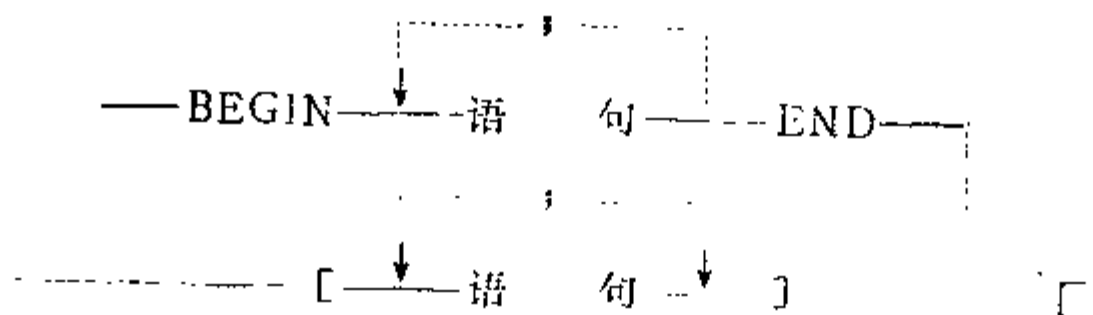
PASCAL语言中，支持这三种结构的语句，包括复合语句（顺序结构），如果语句和情况语句（分支结构），当语句、直到语句和循环语句（循环结构）。它们都属于构造型语句。除此之外，属于简单语句的还有转移语句。这个语句本身完成的是无条件转移，但就其实质看，它必然要与其它语句合在一起，实现按某种条件进行的流程控制。MS PASCAL扩充的BREAK、CYCLE和RETURN则属于GOTO语句的特定用法。

3.1 顺序结构程序设计

在任何一个“正常”的程序中，或在它的一个不是太短的功能程序段中，顺序执行的语句序列所占的比例总是比较高的。因此，顺序结构是程序设计中的基础。

在顺序结构中，除了经常使用赋值语句外，还大量使用复合语句。

复合语句的格式如下：



其中BEGIN和END是复合语句的标志，是PASCAL语言的保留关键字。

BEGIN和END之间是用分号隔开的多个语句，我们称它们为复合语句的成分语句或子语句。成分语句之间用语句分隔符分隔开来，复合语句和后续语句之间也用分号隔开。

复合语句内的成分语句是顺序执行的，而把它们组织在一起，从PASCAL语言的语法规则看，则被作为单个语句进行处理。

复合语句主要用作为其它几种复合语句的子语句。在那些地方，PASCAL的语法要求这些子语句必须是单个语句的形式，当子语句要完成的功能比较复杂，必须由连续的多个语句才能完成时，复合语句就成为解决这一矛盾的必要手段，此时必须把这连续的多个语句组织为完整的一个复合语句，以保证这些语句能顺序执行；当把复合语句用作为分支结构与循环结构中的子语句时，它能保证分支与循环的正确执行。FORTRAN或BASIC语言都缺乏这一功能，因此免不了要用许多转移语句。

标准PASCAL中的复合语句，只能用保留字BEGIN和END“夹”起一个语句序列给出。MS PASCAL允许用[和]代替复合语句中的BEGIN和END。

3.2 分支结构程序设计

在程序设计过程中，常常遇到需要按不同条件去选择不同的程序段完成不同处理的情况。此时，顺序执行结构就无法解决这个问题，而必须采用分支结构。PASCAL语言中，支持这一分支结构的，主要有IF和CASE两个语句，也可以使用GOTO语句。

一、IF（如果）语句

如果语句规定，在一定条件下执行某一成分语句，否则执行另一成分语句，或者不执行任何成分语句。

如果语句有两种形式，其格式如下：

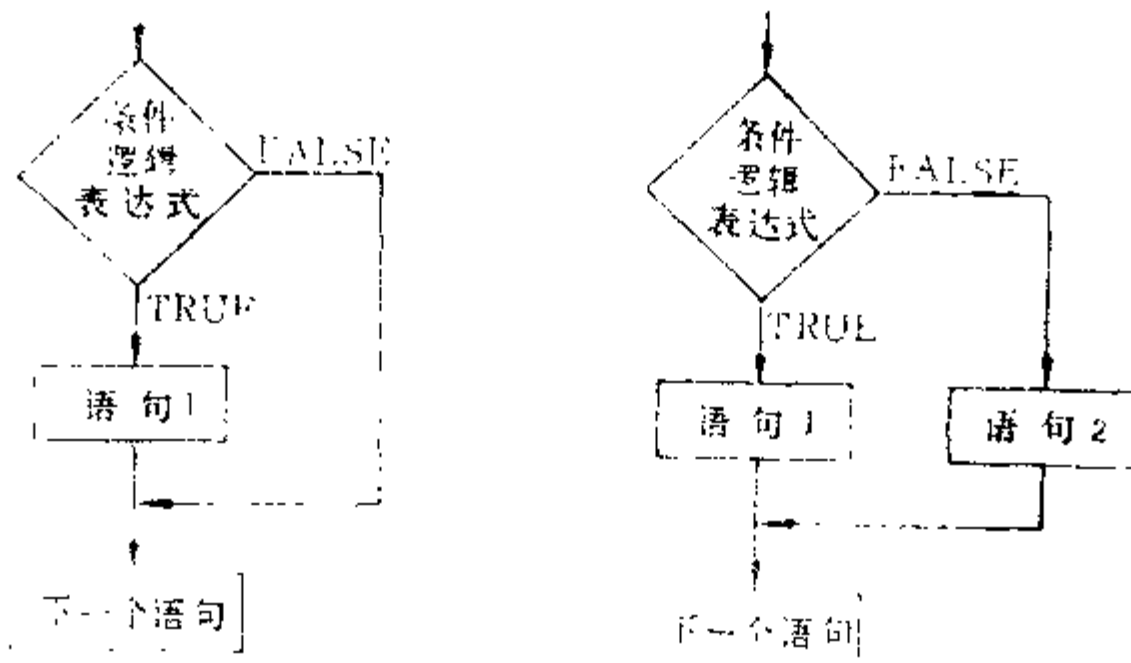
——IF——条件——THEN——语句1——□
或 ——IF——条件——THEN——语句1—— —ELSE——语句2—— □

其中：

IF，THEN和ELSE是如果语句的标志，都是保留关键字。

条件是布尔表达式。可以认为IF语句的第一种形式为第二种形式的一个特例。

如果语句在执行时，其流程为分支结构。流程图如下所示，



图的左边部分描述的是无 ELSE 部分的如果语句的流程图；图的右边描述的是有 ELSE 部分的如果语句的流程图。

如果语句的两种形式，其共同点在于都必须判断条件。在条件成立时，即布尔表达式计算结果为真时，都必须执行一个成分语句 1。它们的主要区别在于，在条件不成立时，第一种形式不执行任何成分语句，直接转去执行下一个语句，而第二种形式必须先执行另一个成分语句 2，然后才转去执行下一个语句。

讨论一个分支结构程序设计的例子。这个程序用来解 $AX^2 + BX + C = 0$ 这个一元二次方程。假定 A、B、C 的值都从终端键盘输入并选为整型量，其值都不为零。具体程序如下：

```
PROGRAM USEIF (INPUT, OUTPUT),
VAR
  A, B, C, D: INTEGER,
  X1, X2: REAL,
BEGIN
  WRITE ('Enter the values of A, B, C: ');
  READLN (A, B, C);
  D = B * B - 4 * A * C;
  IF D >= 0
    THEN
      BEGIN
        X1 := (-B + SQRT(D)) / 2 / A;
        X2 := (-B - SQRT(D)) / 2 / A;
        WRITELN('X1=', X1:10:4, ' X2=', X2:10:4);
      END
    ELSE WRITELN ('B * B - 4 * A * C < 0, No solution');
  WRITELN('PROGRAM COMPLETED')
END.
```

Enter the values of A, B, C: 2 10 5

X1= -0.5635 X2= -4.4365

PROGRAM COMPLETED

Enter the values of A, B, C: 2 5 10

B * B - 4 * A * C < 0, No solution

PROGRAM COMPLETED

在这个程序中，通过判断 $B^2 - 4AC$ 是否小于零这一条件，选择求出该方程两个实根或指明方程无解两种处理。这两种处理一定是二选一，不能都选中，这就是程序设计中分支的概念。最后的输出语句是 IF 的后续语句。

在这个程序中，也可以没有 ELSE 部分。此时，当从终端上输入了 A、B、C 三个数值后，无任何运算结果输出显示，程序就结束运行了。这表明该方程无实数解，这是 IF 语句中没有 ELSE 成分的例子。

顺便再谈一下复合语句在这里的作用。在这个 IF 语句 THEN 之后的部分，要用三

个执行语句。按IF语句的语法规则，THEN之后只能是一个语句，这之后，就是ELSE语句成分，或者是这个IF语句之后的另外一个语句（IF语句无ELSE成分时）。这里正是靠由BEGIN与END包起三个语句而形成一个复合语句，作为IF语句的一个子语句来满足这一语法要求的。如果没有这个BEGIN与END，编译程序就会把向X2赋值开始的语句，作为整个IF语句之后的另外的语句处理，这显然不是我们的要求，而且在遇到后面的ELSE成分时，会指出一个错误，因为IF语句已经结束了，怎么又冒出个ELSE成分呢？在没有ELSE成分时，这最后一个错误是不存在了，但前一个错误依然存在，当 $D < 0$ 时，这个程序运行的结果是错误的。初学者在自己的程序设计里，经常在这种地方犯忘掉使用复合语句的错误。

IF语句本身也可以作为IF语句的子语句，可能形成如下情况：

IF条件1 THEN IF条件2 THEN 语句1 ELSE 语句2;

若不对这个情况做出明确规定，则语句2是属于第一个IF语句的成分，还是第二个IF语句（作为第一个IF语句的子语句）的成分。为解决这里的二义性矛盾，PASCAL语言已明确规定，要把这样的ELSE成分划归为最靠近它的那个（即这里的第二个）IF语句。请在自己的程序设计中，注意使用多个连续的IF语句中的这一特殊问题。

二、CASE（情况）语句

情况语句，是根据用于选择不同情况的一个表达式的计算结果，选择不同子语句加以执行的一条功能较强的语句。

情况语句的标准格式如下：

```

CASE——表达式——OF
    常量1——：——语句1
    常量2——：——语句2
    .....
    常量n——：——语句n
END——[ ]
    
```

其中：

CASE，OF，END是情况语句的标志，是PASCAL语言的保留关键字；

表达式通常是一个变量，一般是整数型变量、字符型变量或枚举型变量，布尔型变量也可以使用。实数型变量不能使用。也可以是一个比较复杂的表达式。

常量是表达式计算结果可以得到的值，因此常量的类型必须与表达式计算结果的数据类型相同。这里的常量都被用作为CASE语句的每个成分语句的语句标号。

END是整个CASE语句的结束标记，它与该CASE语句的最后一个成分语句之间不要有分号。

语句与标号在一起组成CASE语句的一个成分语句，两个相邻的成分语句之间用分号隔开。

为了形象化地表示情况语句，可用下示的展开图格式加以描述：

```

CASE——选择表达式——OF
    情况标号1——：——语句1；
    情况标号2——：——语句2；
    .....
    情况标号n——：——语句n；
END——[ ]
    
```

情况语句的执行过程，是首先计算表达式，再把表达式计算果结，作为成分语句的情况标号，转去执行具有相应标号值的一个成分语句。可以看出，情况语句实现的是从多种可能的情况中，按当前选中的一种情况，执行一种相应的处理。从程序的分支结构看，它实现的是多选一，比实现二选一的如果语句功能要强得多。

情况语句允许几种情况执行同一个成分语句，也允许一种情况执行许多语句，但是，这多个语句必须用BEGIN和END包起来形成复合语句，作为一个完整的成分语句。如果什么情况都不是，则不执行任何成分语句，立即跳出情况语句。

必须指出，情况标号的值是程序执行过程中产生的，不需要在程序的说明部分加以说明。情况标号在情况语句中的次序是任意的，不一定按照从小到大排列。

看一个具体程序的例子。还是解一元二次方程 $AX^2+BX+C=0$ ，但取消A，B，C三个输入值不能为零的限制。根据A，B，C的值是否为零，可以区分出以下多种情况，

情况分类	A、B、C取值情况			方程求解的结果
	A	B	C	
1	0	0	0	输入有错
2	0	0	—	方程无解
3	0	—	0	方程单解
4	—	0	0	方程双解
5	—	0	—	可分为三种情形，见下面部分

对于情况5，又可按 $B*B-4*A*C$ 的值是大于、等于或小于0分为三种情形，编号用5、6、7表示如下，

情形编号	$B*B-4*A*C$ 的值	方程求解结果
5	> 0	方程有两个不等实数解
6	$= 0$	方程有两个相等实数解
7	< 0	方程无实数解

```

下边给出的是该程序的源清单和运行结果：
PROGRAM USECASE(INPUT, OUTPUT),
VAR
  A, B, C, D, CASELAB; INTEGER,
  X1, X2; REAL,
BEGIN
  WRITE('Enter the values of A,B,C: '),
  READLN(A,B,C),

```

```

(* PART 1, DISTINGUISH DIFFERENT CASE *)
IF A = 0
  THEN
    IF B = 0
      THEN
        IF C = 0
          THEN CASELAB := 1
          ELSE CASELAB := 2
        ELSE CASELAB := 3
      ELSE
        IF C = 0
          THEN CASELAB := 4
          ELSE
            BEGIN
              D := B * B - 4 * A * C;
              IF D > 0
                THEN CASELAB := 5
                ELSE
                  IF D = 0
                    THEN CASELAB := 6
                    ELSE CASELAB := 7
                  END;
            END;
          (* PART 2, OUTPUT DIVERS RESULTAT *)
          CASE CASELAB OF
            1 : WRITELN('ERROR IN INPUT');
            2 : WRITELN('NO SOLUTION');
            3 : WRITELN('X =', C/B:10:4);
            4 : WRITELN('X1 =', B/A:10:4, 'X2 =', 0.0:10:4);
            5 : WRITELN('X1 =', (-B + SQRT(D)) / 2/A:10:4,
                      'X2 =', (-B - SQRT(D)) / 2/A:10:4);
            6 : WRITELN('X1, X2 =', -B/2/A:10:4);
            7 : WRITELN('NO REAL SOLUTION');
          END;
          (* OF CASE *)
          WRITELN('PROGRAM COMPLETED')
        END.
Enter the values of A,B,C: 0 0 0
ERROR IN INPUT
PROGRAM COMPLETED
Enter the values of A,B,C: 0 0 1
NO SOLUTION

```

```

PROGRAM COMPLETED
Enter the values of A,B,C: 0 1 0
X= 0.0000
PROGRAM COMPLETED
Enter the values of A,B,C: 0 1 2
X= 2.0000
PROGRAM COMPLETED
Enter the values of A,B,C: 3 1 0
X1= 0.3333 X2= 0.0000
PROGRAM COMPLETED
Enter the values of A,B,C: 2 0 0
X1= 0.0000 X2= 0.0000
PROGRAM COMPLETED
Enter the values of A,B,C: 5 0 5
NO REAL SOLUTION
PROGRAM COMPLETED
Enter the values of A,B,C: 5 0 -10
X1= 1.4142 X2= -1.4142
PROGRAM COMPLETED
Enter the values of A,B,C: 2 10 5
X1= 0.5635 X2= -4.4365
PROGRAM COMPLETED
Enter the values of A,B,C: 2
NO SOLUTION
PROGRAM COMPLETED
Enter the values of A,B,C: 1 2 1
X1, X2= -1.0000
PROGRAM COMPLETED

```

这个程序的关键部分，由一个IF语句和一个CASE语句实现。IF语句用于区分几种不同的情况，CASE语句用于输出相应的结果。

最后补充说明一点，许多PASCAL编译程序都在CASE语句内加了一个扩充的子语句，MS PASCAL就增加了一个OTHERWISE子语句。OTHERWISE在这里当作语句标号使用，代表一组常量，即没在这个CASE语句中明确作为CASE标号使用的表达式的那些计算结果。

在情况语句中包含OTHERWISE功能时，必须注意以下四点：

- (1) 它必须在所有情况标号之后，不允许插在中间，
- (2) 与它前面的那个语句之间，必须有语句分隔符——分号，
- (3) 它后面直接跟语句 $n+1$ ，中间不允许用冒号分开。这一点不同于通常的情况标号。语句 $n+1$ 可以是简单语句或构造型语句，
- (4) 在语句 $n+1$ 与END之间不允许用分号，这一点不同于一般的情况语句。

在一个情况语句中，情况标号必须是互不相同的，不允许有二义性。也就是说，在一个情况语句中，任何一个情况标号只能出现一次。

三、标号说明与GOTO（转移）语句

在各种算法语言中，几乎都有转移语句。然而在PASCAL语言中，为了程序动态运行和静态结构相一致，不提倡使用转移语句，而且，程序设计得好，也确实能做到不使用转移语句。

转移语句的功能是改变程序执行的顺序，跳过程序的某一部分转去执行另一部分，或者返回前面已经执行过的某一个语句使之重复执行。它通常作为条件语句的成分语句。

PASCAL语言的转移语句的一般格式如下：

——GOTO——语句标号——□

其中：

GOTO是转移语句的标志，是一个保留关键字。请注意，GOTO不允许分开写成GO TO。这一点不同于其它算法语言。语句标号表示程序转移去执行的那个语句的标号。

在PASCAL语言中，凡是程序执行部分使用的语句标号，必须在程序说明部分进行标号说明。标号说明的格式如下：

——LABEL——语句标号——□

其中LABEL是标号说明的标志，它是PASCAL语言的一个保留关键字。

标准PASCAL语言规定，语句标号是整数，允许从1到9999。一般是由小到大排列，可以是不连续的排列顺序。在MS PASCAL语言中规定，也允许把标识符用作语句标号。

在程序执行部分，语句标号后面必须有语句跟随，表示转移到这儿干什么。标号语句的格式如下：

——语句标号——：——语句——□

语句标号必须在程序说明部分已经说明，标号和语句之间必须有冒号。

语句可以是任何语句，基本语句和构造型语句都可以，也可以用空语句。

看两个小程序的例子。其功能都是读入一个实型量，再计算并输出它的平方根值。

```
PROGRAM USEGOTO (INPUT, OUTPUT);
LABEL
    10;
VAR
    X, Y: REAL;
BEGIN
    READLN(X);
    IF X < 0
    THEN
        BEGIN
            WRITELN ('CANNOT SOLVE');
```

```

        GOTO 10
    END;
    Y:=SQRT(X);
    WRITELN('Y=',Y:6:2);
10:
END.
-6.1
CANNOT SOLVE
6.1
Y= 2.47

```

这里的标号10后跟随的是一个空语句。

这个程序运行结果是正确的。而下面一个程序则有运行错误。

```

    PROGRAM USEGOTO(INPUT, OUTPUT);
    LABEL
        15;
    VAR
        X, Y:REAL;
    BEGIN
        READLN(X);
        IF X<0
            THEN GOTO 15;
        Y:=SQRT(X);
        WRITELN('Y=',Y:6:2);
15:
        WRITELN('CANNOT SOLVE')
    END.
-6.1
CANNOT SOLVE
6.1
Y= 2.47
CANNOT SOLVE

```

这个程序对 $X < 0$ 的情况，运行结果是正确的。但当 $X \geq 0$ 时，在完成计算Y值并输出Y的结果之后，接着又执行了标号为15的语句，出现运行错误。这是 PASCAL 语言的一些初学者，在试图用 GOTO 语句完成分支控制中常犯的一个错误，即只考虑到问题的一个方面，却忽略了对另外一个方面的影响。

克服上面这个错误，可以再多设置一个标号 20，在输出Y的语句之后加一个 GOTO 20 语句；把标号20，写在程序的 END 之前。

对 MS PASCAL，可以用标识符作为语句标号。例如，可以把上个程序中 LABEL 15 变为 LABEL LAB，并把 GOTO 15 变为 GOTO LAB，把语句标号 15，变成 LAB:，程序运行的结果将不变。

本节小结

分支结构，是程序设计常用的三种结构中比较重要的一种，比顺序结构难掌握，比循环结构更常用，对程序的可读性与程序控制逻辑上的清晰、正确性，都有直接的影响。

在进行分支结构设计时，要把握好每个程序分支的功能划分、各有关分支间的关系，并选择好控制分支转移的条件与时机，尽量避免那些繁锁的转来转去的多余操作。

在支持分支结构的三个语句中，CASE语句的功能最强，出错机会相对少一点。在实现多选一分支控制时，尽量用CASE语句，而不是用多个IF语句或一个结构非常复杂的多层IF语句实现。CASE的子语句还可以是CASE语句，这进一步增强了它的功能。

IF语句在实现二选一的分支结构中最有用。当由多层的IF子语句构成复杂的IF语句时，避免各部分之间的重复和矛盾现象，明确每一部分所反映的真正条件（如USECASE程序中开头部分的IF语句），找出忘记处理的方面或别的一些错误是十分重要的。

GOTO语句虽然也可以实现分支控制，但以尽量少用为好。编译程序对GOTO语句进行的错误检查是不太严格的，主要靠用户自行解决。可以考虑用在转移条件非常简单明确，后续处理可以说得很清楚的地方。要尽量避免用GOTO完成循环执行的情况，要尽量避免用多个GOTO在程序中彼此转来转去的情况。

3.3 循环结构程序设计

循环结构是程序设计中常用的手段，它是按指定的次数，或某个控制条件，让一段程序循环执行。PASCAL语言中，支持这一结构的有三种语句，它们控制循环的办法各不相同。

FOR语句，是按指定的次数控制循环。

WHILE语句，是按选定的条件控制循环。在进入循环之前，先检查控制条件，条件成立，就执行成分语句，否则结束循环。

REPEAT语句，也是按选定的条件控制循环，但它是先执行一次成分语句，然后检查控制条件，条件成立，结束循环，否则继续循环。

下面分三小节介绍这三种语句的格式与使用方法。

一、FOR（循环）语句

在PASCAL语言中，循环语句规定，当控制变量在给定的范围内变化时，它重复执行成分语句。重复的次数是由语句的控制变量的初值与终值决定的，一般与成分语句的执行情况无关。

循环语句有两种格式，如下图所示：

----- TO -----

——FOR——标识符——:=——初值——DOWNTO——终值——DO——语句——□

依据循环语句中用的是TO还是DOWNTO，把循环语句区分为循环控制变量是递增还是递减两种不同类型的循环语句。

其中：标识符通常是一个变量名。

FOR、TO、DOWNTO和DO是循环语句的标志，都是保留关键字。

变量名作为控制变量使用，它必须属于有序的数据类型、常用整数类型和字符类型变量。也可以用枚举类型和子界类型变量。但不允许用实数类型变量。

初值和终值一般为算术表达式。它们的运算结果必须与控制变量属于同一种类型。

语句为成分语句。成分语句可以是复合语句。成分语句组成循环体。

$:=$ 是赋值运算符。

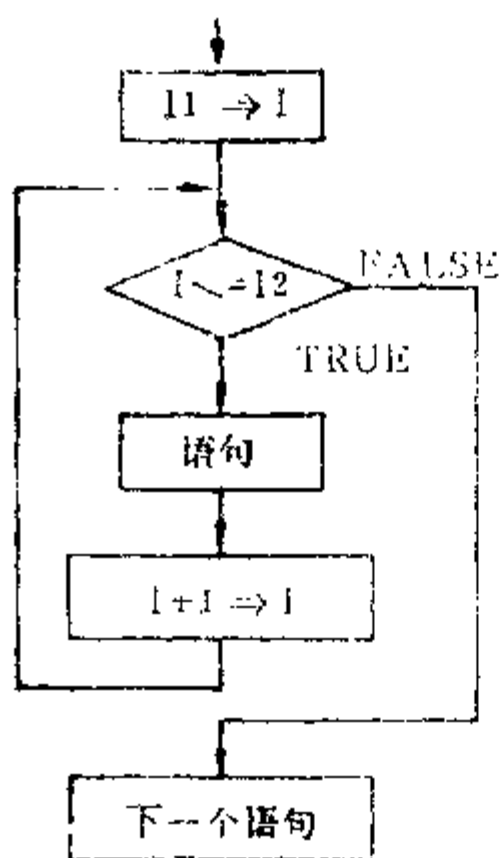
TO是控制变量递增型循环语句的标志，它要求终值大于初值，DOWNTO是控制变量递减型循环语句的标志，它要求终值小于初值。它们都是保留关键字。

在PASCAL语言中，循环语句没有步长这一项。如果控制变量属于整数类型，那么递增型的步长为 $+1$ ，递减型的步长为 -1 。对于其它类型的控制变量，其步长要由类型确定，情况与上面类似。

下面以整数类型控制变量为例，画出以下两个语句的流程图：

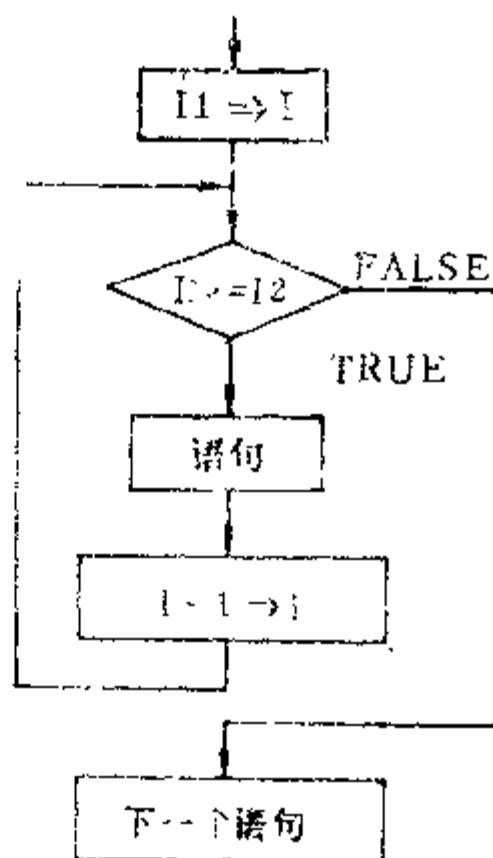
FOR $I:=I_1$ TO I_2 DO 语句

或 FOR $I:=I_1$ DOWNTO I_2 DO 语句



(a)

(a) 递增型



(b)

(b) 递减型

图 (a) 是递增型循环语句流程图， $I+1 \rightarrow I$ 的工作是由机器自动完成的。因此，不需要在程序中书写赋值语句 $I := I + 1$ 。否则，是一个错误。应该说，在 FOR 语句循环体内人为修改循环控制变量是不允许的。

图 (b) 是递减型循环语句流程图。要修改控制条件为 $I \geq I_2$ 同时将 $I+1 \rightarrow I$ 改为 $I-1 \rightarrow I$ ，其余与递增型循环语句相同。

从上面的递增型流程图中，可以看出循环语句的执行过程如下：

(1) 将初值赋给控制变量；

- (2) 判断循环条件是否成立, 如果条件不成立, 则转步骤 6;
- (3) 条件成立时执行循环体;
- (4) 修改控制变量;
- (5) 返回步骤 2;
- (6) 结束循环过程, 执行下一个语句。

必须指出, PASCAL 语言的循环语句不同于某些算法语言的循环语句。它是先判断循环条件是否成立, 后执行循环体。这一点与当语句非常类似。

对于不同的循环条件, 循环语句的执行情况也不相同。对于递增型循环语句有三点:

- (1) 如果 $I_2 < I_1$, 则循环条件不成立。程序运行时不执行循环体, 立即转去执行下一个语句。
- (2) 如果 $I_2 = I_1$, 则循环条件成立。程序运行时只执行一次循环体。
- (3) 如果 $I_2 > I_1$, 则循环条件成立。程序运行时重复执行循环体, 重复次数为 $I_2 - I_1 + 1$ 。

具体程序的实例在后面章节中给出。

二、WHILE (当) 语句

当语句规定, 当条件成立时, 重复执行成分语句。重复的次数取决于成分语句的执行情况。

当语句的一般格式如下,

——WHILE——条件——DO——语句——□

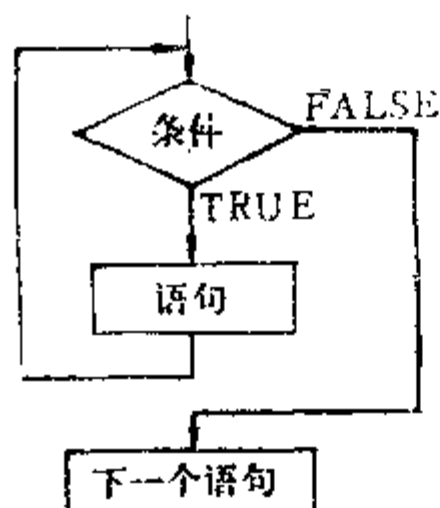
其中,

WHILE, DO 是当语句的标志, 是 PASCAL 语言的保留关键字;

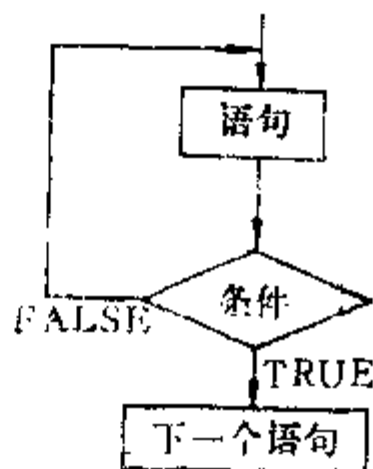
条件为布尔表达式;

语句为成分语句。成分语句可以是复合语句。

当语句的流程图如下:



直到语句的流程图如下:



从当语句的流程图可以看出, 当语句的主要特点如下:

- (1) 首先判断条件, 决定是否要执行成分语句。条件一开始就不成立, 则不执行成

分语句。

(2) 条件成立时, 继续执行成分语句。在正常使用的当语句中, 成分语句的执行结果应该改变这一条件, 否则会出现死循环, 要警惕这种设计上的错误。条件不成立时, 则执行下一个语句。

三、REPEAT(直到)语句

PASCAL 语言规定, 直到语句是使程序重复执行它的成分语句, 直到条件成立时为止。重复的次数取决于成分语句的执行情况。

直到语句的一般格式如下:

——REPEAT —— 语句 · —— UNTIL —— 条件 —— □

其中:

REPEAT和UNTIL 是直到语句的标志, 是PASCAL语言的保留关键字;

条件必须为布尔表达式;

语句为成分语句。在成分语句超过一个时, 也不必构成复合语句。

直到语句的特点如下:

(1) 不管情况如何, 首先执行成分语句, 然后判断条件。

(2) 条件不成立时, 继续执行成分语句。执行成分语句的结果, 应能改变这一条件, 否则就会出现死循环。这与当语句情况类似, 应尽量避免。条件成立, 执行下一个语句。

四、多层循环和提前退出循环问题

如果在循环体中包含另一个循环语句, 则称为二重循环。在第二层循环中又包含一个小的循环, 则称为三重循环。……继续一层套一层, 则形成多重循环。

关于多重循环的嵌套问题, 与其它算法语言一样, PASCAL 语言也有一些规定。用户在程序设计时, 必须遵循如下规定:

1. 外循环体一般包括内循环及其它语句, 但是内循环必须完全在外循环体之内, 内外循环不得相互骑跨。

2. 内循环在外循环体中的位置是任意的, 要尽可能避免在多重循环程序段间使用GOTO语句。

3. 内外循环的控制变量不能同名, 控制循环的条件不要彼此冲突。

4. 外循环体允许包括几个内循环, 这些并列的内循环允许用相同的控制变量。

在循环语句执行的过程中, 如果满足了一定的条件, 不希望继续循环, 则可以立即退出循环以节省运行时间。

标准的PASCAL 语言, 没有专用的退出循环的语句。在一般情况下, 可以通过在条件语句的成分语句中使用转移语句的办法来解决问题。例如:

```

FOR I:=1 TO 100 DO
  BEGIN
    J:=2*I+50;
    IF J>130
      THEN GOTO 10
  END;

```

10:语句;

在这一段程序中,只要满足 $J>130$ 这一条件,就马上结束循环,跳到循环体后的语句去执行了。这种由用户用GOTO语句解决问题的办法有一定出错机会,为此,不少版本的PASCAL语言,都扩充了一个专门用于提前结束循环的语句。一般情况下,它总是用作条件语句的成分语句,单独使用是没道理的。MS PASCAL语言就扩充了一个BREAK语句,此时就可以把上面例子中的GOTO 10换成BREAK,语句标号10也可以取消,使上一段程序变成如下形式:

```

FOR I:=1 TO 100 DO
  [ J:=2*I+50;
    IF J>130 THEN BREAK ];

```

语句;

这里还用 [和] 替代了复合语句中的BEGIN和END。MS PASCAL是支持这种替代用法的。

这个BREAK语句一般具有下述特点:

- (1) 它仅能用在循环语句、当语句和直到语句这三种重复语句中。
- (2) BREAK语句执行时,只能退出循环。或广义地讲,不再重复。一般情况下并不退出整个程序。
- (3) 在多重循环的情况下,BREAK 语句退出的是直接使用它的那一层循环,不是退出整个的循环。

3.4 MS PASCAL 语言在分支与循环控制结构中实现的扩展功能

MS PASCAL语言对IF语句、WHILE语句和REPEAT语句的控制条件进行处理时,扩充了用AND THEN 和OR ELSE指明分析次序的功能,并扩充了BREAK和CYCLE两个语句。

必须强调的是,用AND THEN和ORELSE来指明分析一个逻辑表达式中处理的次序,只适用于对 IF 语句、WHILE 语句和 REPEAT 语句中的条件表达式的分析,而不能用于对其它用法的逻辑表达式的分析处理。

一、分析条件表达式时的处理次序

下面介绍 MS PASCAL 语言在分析与使用分支控制结构和循环控制结构的控制条件时,对那里的条件表达式处理中的扩展功能。这种扩展表现为:

(1) 用AND THEN明确两个逻辑值的“与”运算中的次序关系。例如:

```
IF A>=B AND THEN C<D THEN...
```

```
WHILE A=0.5 AND THEN C>50 DO...
```

这意味着当控制条件表达式中的第一个关系运算的结果为真时(即 $A \geq B$ 为真,或 $A=0.5$ 为真),才会接下去处理AND THEN之后的第二个关系表达式(即 $C < D$,或 $C > 50$)。否则,将直接结束条件表达式分析,并确认条件表达式的结果为假。

(2) 用OR ELSE明确两个逻辑值的或运算中的次序关系。例如:

```
IF A>=B OR ELSE C<D THEN...
```

```
REPEAT
```

```
.....
```

```
UNTIL A=0.5 OR ELSE C>50
```

这意味着当控制条件表达式中的第一个关系运算的结果为假时(即 $A \geq B$ 为假,或 $A=0.5$ 为假),才会接下去处理OR ELSE后面给出的第二个关系表达式,否则,将直接结束条件表达式分析,并确认条件表达式的结果为真。

这样做的好处是明显的。第一,当第二个关系表达式的分析能否正确进行要以第一个关系表达式的结果为前提时,先分析完第一个,再按得到的结果决定是否还要讲行第二个关系表达式的分析,才能保证程序肯定能正确运行。例如,后面讲到数组应用时,就可以用第一个关系表达式先检查数组下标变量的值是否正确,只有在其值正确时,才按这个下标值去访问数组相应元素的值,这样就一定不会出现数组下标越界的错误。其次,这样做在某些条件下可以节省得到条件表达式的执行结果的运算时间,即只分析了条件表达式中的第一个关系运算的结果,就已经确定了条件表达式的最终结果,也就不必再分析其后面的内容了。

二、BREAK与CYCLE语句

对循环控制结构,MS PASCAL语言还给出了BREAK和CYCLE两种扩充语句。从功能上看,它们是GOTO语句在循环控制结构中的特定用法,是MS PASCAL语言实现的定型了的GOTO语句。

BREAK语句用于提前结束WHILE、REPEAT和FOR语句的循环执行。CYCLE语句则用于使FOR语句的循环控制变量变为它的下一个后续值,并让FOR语句从头接着运行。这样做的原因是为了使程序看起来更清楚,又能减少用户直接使用GOTO语句可能带来的错误。

BREAK和CYCLE语句都有两种格式。一种格式是它们跟有一个循环语句的语句标号,此时将退出或重新运行的是由标号标明的循环语句。这在有多重循环的情况下,会表示得更加清楚。另一种格式是不跟语句标号的。此时是退出或重新运行它们所在的循环语句,在多重循环的情况下,退出或重新运行的是它所在的最内层的一重循环,而不是整个的循环。

看如下几个用法实例:

```
OUTER FOR I:=1 TO N1 DO
```

```
INNER FOR J:=1 TO N2 DO
```

```

IF A[I, J] = TARGET
THEN BREAK OUTER,

```

BREAK后跟有语言标号OUTER，是外层循环语句的标号，故BREAK结束该外层循环，也就是整个循环过程。

若BREAK后不跟标号，或跟内层循环语句的标号INNER，则BREAK将只结束本次内层循环，若I的值尚未达到N1，该循环将从下一个I值开始新一轮循环过程。

```

SEARCH: WHILE I <= ITOP DO
    IF PILE[I] = TARGET
    THEN BREAK SEARCH
    ELSE I := I + 1;

```

因为此处只有一重循环，BREAK后跟不跟语句标号SEARCH，对该WHILE语句的执行不产生任何影响。当BREAK后不跟语句标号，其它处又不用GOTO SEARCH语句时，就可以取消SEARCH标号。

```

FOR I := 1 TO N DO
    IF NEXT[I] = NIL THEN BREAK;

```

这是BREAK不带语句标号的一个例子。

```

CLIMB: WHILE NOT ITEM^. LEAF DO
    BEGIN
        IF ITEM^. LEFT < > NIL
        THEN [ ITEM := ITEM^. LEFT,
              CYCLE CLIMB ];
        IF ITEM^. RIGHT < > NIL
        THEN [ ITEM := ITEM^. RIGHT,
              CYCLE CLIMB ];
        WRITE ('Very strange node');
        BREAK CLIMB;
    END;

```

这是应用BREAK和CYCLE语句控制WHILE语句运行过程的一个例子。由于此处用到了指针类型变量，前边的三个小例子中却用到了数组变量，都是尚未学到的内容。初次接触PASCAL程序设计的人员，若在这些地方读不懂，可在学习完第四章之后再回过头来重读这里的内容。

3.5 循环结构程序设计举例

前几节分别介绍了MS PASCAL语言中支持循环结构的五种语句。我们把程序设计的实例集中放在这一小节中，是为了更便于比较这几种语句在功能与用法上的同异之处，加深读者对这些问题的理解程度。

由于我们目前只学习了简单类型的数据的用法，构造类型的数据尚未学到，所以这里给出的都是一些很简单的小程序。构造型语句常常用来处理构造类型的数据结构。结构设计，不仅要求所用语言有良好的语句支持程序的控制结构，也要求有良好的构造数据

类型支持常用的数据结构。

第一个程序是应用FOR语句求出九九乘法表的乘积的累计和结果的一个例子。这^①用了IF语句和GOTO语句，程序中的逻辑关系不那么直观。程序的输出为1155。

```
PROGRAM SUM (OUTPUT),
LABEL
  1,
VAR
  SUM, I, J: INTEGER,
BEGIN
  SUM:=0; J:=9;
1:
  FOR I:=J DOWNTO 1 DO
    SUM:=SUM+J*I,
  IF J<>1
  THEN
    BEGIN
      J:=J-1; GOTO 1
    END,
  WRITELN ('SUM=',SUM:5)
END.
SUM=1155
```

第二个程序改用FOR语句的二重循环结构，语句更为简洁，逻辑关系也更直观。能用循环次数控制的运算操作最好用FOR语句实现。

```
PROGRAM SUN (OUTPUT),
VAR
  SUM, I, J: INTEGER,
BEGIN
  SUM:=0;
  FOR I:=1 TO 9 DO
    FOR J:=1 TO I DO SUM:=SUM+J*I,
  WRITELN ('SUM=',SUM:5)
END.
SUM=1155
```

第三个程序应用了FOR语句的三重循环结构。程序的功能是求出用一定的钱数，购买数量一定的三种价格的物品的全部可能组合情况。

```
PROGRAM BUYFOOD (INPUT, OUTPUT),
CONST
  COST 1:=4,
  COST 2:=3,
  COST 3:=1,
```

```

VAR
    I, J, L, MONEY, TOTAL:INTEGER;
    SUCCESS,BOOLEAN;
BEGIN
    SUCCESS:=FALSE;
    WRITE('ENTER THE VALUES OF MONEY AND TOTAL:');
    READLN(MONEY, TOTAL);
    IF (MONEY<=COST1 * TOTAL) AND (MONEY>=COST3
        * TOTAL)
    THEN
        FOR I:=0 TO MONEY DIV COST1 DO
            FOR J:=0 TO MONEY DIV COST2-I DO
                FOR L:=0 TO TOTAL-I-J DO
                    IF (I * COST1+ J * COST2+ L * COST3=MONEY) AND
                        (I+J+L=TOTAL) THEN
                        BEGIN
                            WRITELN(I, J, L); SUCCESS:=TRUE
                        END;
                IF NOT SUCCESS
                    THEN WRITELN('NOT SUCCESS!')
            END.
ENTER THE VALUES OF MONEY AND TOTAL,50 60
NOT SUCCESS!
ENTER THE VALUES OF MONEY AND TOTAL,30      9
          3          6          0
          5          3          1
          7          0          2

```

这个程序中钱数与欲购物品总数MONEY和TOTAL是由终端送入的。三种价格在常量定义部分给出。求出全部可能组合情况采用的算法是试探法，即把全部能出现的情况都进行一番检查，找出购物总数与总钱数都满足要求的那些组合。

这个程序看起来简单，但编写得是否合理对程序的质量有重大影响。

第一，这里用了三重循环，怎么决定每一重的循环次数非常关键。不仔细考虑，很容易写成：

```

FOR I:=0 TO TOTAL DO
FOR J:=0 TO TOTAL DO
FOR L:=0 TO TOTAL DO

```

这种写法逻辑上是说得通的，运算结果也正确，但程序的执行时间却浪费极大。当TOTAL为500时，循环体部分将执行 $500 \times 500 \times 500$ 次，时间相当长。实际上用不着这

样做,对I来讲,最大值能取 $MONEY \div COST1$, J能取 $MONEY \div COST2 - I$, L能取 $TOTAL - I - J$ 。

第二,对有些情况,根本不必进行试探。如给出的钱数都用来买最贵的物品,买得的数量比欲购量都要大,或者都用来买最贱的物品,购得的数量都比欲购量要小,肯定找不到任何组合。这就是READLN语句之后的那个IF语句的作用。

第三,如果进一步推敲,这个程序还可以改进。因为三层循环中的最内层循环是多余的。I和J一定下来,L应为 $TOTAL - I - J$,不必试探,此时要把循环体内的IF语句改为:

IF $I * COST1 + J * COST2 + (TOTAL - I - J) * COST3 = MONEY$

最后要说明一点,对某些变量要给初值,如这里的SUCCESS。有时不给初值程序运行结果也可能正确,但这并没有绝对把握,也是一种不好的程序设计习惯。

第四个程序实现的是在终端屏幕上显示九行九列的九九乘法口诀表。这个程序的目的是提示怎样控制数据和有关标题的输出格式。

```
PROGRAM MULTIPLE (OUTPUT);
VAR
  I, J:INTEGER;
BEGIN
  WRITE (' ':6);
  FOR I:=1 TO 9 DO WRITE (I:6);
  WRITELN;
  WRITELN;
  FOR I:=1 TO 9 DO
    BEGIN
      WRITE (I:6);
      FOR J:=1 TO 9 DO
        WRITE (J * I:6);
      WRITELN
    END
  END.
END.
```

	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9
2	2	4	6	8	10	12	14	16	18
3	3	6	9	12	15	18	21	24	27
4	4	8	12	16	20	24	28	32	36
5	5	10	15	20	25	30	35	40	45
6	6	12	18	24	30	36	42	48	54
7	7	14	21	28	35	42	49	56	63
8	8	16	24	32	40	48	56	64	72
9	9	18	27	36	45	54	63	72	81

第五个和第六个程序分别是应用 WHILE语句和 REPEAT 语句的例子。程序的功能

都是从终端输入一串字符，统计字符问号之前（包括问号本身在内）的英文字母、数字字符和其它字符的个数。请注意，不同的计算机系统上的 PASCAL 语言，在处理终端字符输入时，对头一个字符的输入的处理方式可能不一样，请参阅有关手册，一般可以在读入第一个字符之前先用一个 READLN 语句。两个程序的运行结果有差异，哪一个错了？

```

PROGRAM STATISTICS (INPUT, OUTPUT);
VAR
    IALPHA, INUMB, OTHERS: INTEGER;
    CH: CHAR;
BEGIN
    IALPHA := 0; INUMB := 0; OTHERS := 0;
    READ (CH);
    WHILE CH < '>' DO
        BEGIN
            IF (CH <='Z') AND (CH >='A')
                THEN IALPHA := IALPHA + 1
            ELSE
                IF (CH <='9') AND (CH >='0')
                    THEN INUMB := INUMB + 1
                ELSE OTHERS := OTHERS + 1;
            READ (CH)
        END;
    READLN;
    WRITELN ('IALPHA=', IALPHA, 5);
    WRITELN ('INUMB=', INUMB, 5);
    WRITELN ('OTHERS=', OTHERS, 5)
END.

```

```

1234ASDF; '':.] 0 [) (* & -5678 VBNM?ASDFG12.
IALPHA=      8
INUMB  =      9
OTHERS=     11

```

```

PROGRAM STATISTICS (INPUT, OUTPUT);
VAR
    IALPHA, INUMB, OTHERS: INTEGER;
    CH: CHAR;
BEGIN
    IALPHA := 0; INUMB := 0; OTHERS := 0;
    REPEAT
        READ (CH);
        IF (CH <='Z') AND (CH >='A')
            THEN IALPHA := IALPHA + 1

```

```

        ELSE
        IF (CH<=' 9') AND (CH>=' 0')
        THEN INUMB:=INUMB+1
        ELSE OTHERS:=OTHERS+1;
UNTIL CH='?';
READLN;
WRITELN('ALPHA=',ALPHA:5);
WRITELN(',INUMB=',INUMB:5);
WRITELN(',OTHERS=',OTHERS:5)
END.

```

```

1234ASDF; '':] 0 [) (* & -5678VBNM)? ASDFG12.
ALPHA=      8
INUMB  =     9
OTHERS=    12

```

第七个和第八个程序是属于设计上不那么十分好的程序。第七个程序与本节的第一与第二个程序功能相同，比第一个程序好，比第二个程序差，应该用次数控制的一个循环是用WHILE语句实现的。

```

PROGRAM SUM (INPUT, OUTPUT);
VAR
    SUM, I, J, INTEGER;
BEGIN
    SUM:=0;    J:=9;
    WHILE J>0 DO
        BEGIN
            FOR I:=J DOWNT0 1 DO
                SUM:=SUM+J*I;
                J:=J-1
            END;
        WRITELN ('SUM=',SUM:5)
    END.

```

第八个程序的功能是计算20年内国内经济总产值要翻两番，则应每年平均递增率为多少。这个程序的不足之处在于把应该用REPEAT实现的一个循环用一个IF和一个GOTO语句来完成的，程序不够简炼，逻辑关系还不够清楚。请读者把这个程序改得更好。

```

PROGRAM CALCUL (OUTPUT);
LABEL
    1;
VAR
    R, R1, X, REAL;
    I, INTEGER;
BEGIN

```

```

    R1:=4.00; X:=0.0010;
1:
    R:=1.0;
    FOR I:=1 TO 20 DO
        R:=R*(1.00+X);
    IF R1>R THEN
        BEGIN
            X:=X+0.0005; GOTO 1
        END;
    WRITELN ('R=',R:6:3,'X=',X:6:3)
END.
R=4.017 X= 0.072

```

习 题

1. 结构程序设计中的三种基本程序结构都是什么结构? 用这三种结构的组合与嵌套等就能实现各种算法吗?

2. PASCAL 语言的空语句有什么用处? 能在这一章中给出的几个程序中找出几处用到空语句的地方吗?

3. 复合语句的作用是什么? 为什么用PASCAL语言进行程序设计的过程中可以不使用GOTO语句?

4. 实现分支结构的三种语句各自用在什么条件下更合理一些?

5. 实现循环结构的三种语句各自用在什么条件下更合理一些?

6. 请把本章第二节中给出的程序USEIF改成仍用IF语句实现的, 按 $D>0$, $D=0$ 和 $D<0$ 三种情况, 分别输出该方程的两个实解, 一个实解与无实数解三种结果的程序。

7. 请把本章第二节中给出的程序USGOTO1 改正确。一种方法是再多用个 GOTO 语句, 另一种方法是取消该程序中原来用到的 GOTO 语句, 体会根本不用 GOTO 语句进行 PASCAL 程序设计的可能性和可能得到的好处。

8. 比较本章第三节给出的三个同样功能的小程序SUM, 说明选用正确合用的语句进行程序设计的必要性, 体会“好”的程序要达到什么标准(例如程序设计的简明性, 逻辑关系的清晰性, 程序的可读性、易修改性等等)。

9. 估算本章第三节给出的程序BUYFOOD用三重各为TOTAL次的循环与用最巧的办法实现计算所用时间之比。

10. 如何把程序 MULTYP LE改成只输出计算结果的左下半部分(即每行只输出与行号相同个数的计算结果)的一个程序。

11. 把本章最后给出的一个程序CALCUT改用REPEAT语句实现。

12. MS PASCAL语言对IF语句、WHILE语句和REPEAT语句中的条件表达式的分析办法进行了什么改善, 它的主要好处表现在哪里?

13. MS PASCAL 语言对程序中循环结构和分支结构的控制扩充了哪几种语句? 用这几种语句比程序人员直接用GOTO语句实现流程控制有什么好处?

第四章 构造型数据类型

在第二章，已经讨论了PASCAL语言的简单数据类型。属于简单类型的数据，由一系列基本数据值组成，并直接通过变量名来访问它们。

本章介绍PASCAL语言的构造数据类型。属于构造类型的数据，由多个元素数据按一定规则组合而成。元素数据，又称成分数据或基数据，它是构成构造型数据的基础元素，它既可以是简单类型，也允许为构造类型。一定的规则，体现为构造不同数据类型的方法不同。一般情况下，仅仅使用变量名是难以访问构造类型的数据的，往往还要再用一个与构造方法有关的附加信息。

基本的构造类型有四种，它们是数组类型、记录类型、集合类型和文件类型。它们还可以按一定规则进行复合和嵌套，组成更复杂的数据结构。在分节详细讲解这四种基本的构造类型之前，先用表格形式扼要描述说明与使用属于这四种类型的变量的有关的内容：

基本数据结构

构造类型	数 组	记 录	集 合	文 件
说明变量的方式	$a: \text{array}[1]$ of T_0	$r: \text{record}$ $S_1: T_1$ $S_2: T_2$... $S_n: T_n$ end	$S: \text{set of } T_0$	$f: \text{file}$ of T_0
选择数据的一个成分的选择符	$a[i] \ (i \in I)$	$r.S$ $(S \in \{S_1, S_2, \dots, S_n\})$	无	f^{\wedge}
访问数据一个成分的方法	具有可计算的下标值 i 的选择符	带有被说明的成分名 S 的选择符	使用关系运算符 \in 的成分判别进行检验	$\text{get}(f)$ $\text{put}(f)$
成分类型	都相同，均为 T_0	可以不同，分别为 T_1, T_2, \dots, T_n	都相同，均为 T_0	都相同，均为 T_0
基 数	$\text{card}(T_0)^{\text{card}(I)}$	$\prod_{i=1}^n \text{Ccard}(T_i)$	$2^{\text{card}(T_0)}$	长度可变

下面我们给出程序结构和数据结构的对应情况。

结 构 类 型	程 序 语 句	数 据 结 构
不可分元素	赋值语句	标准类型
列 举	复合语句	记录类型
重复次数已知	FOR语句	数组类型
选 择	IF 语句	记录变体、类型的并
重复次数未知	WHILE、REPEAT语句	文件类型
递 归	过程语句	递归数据类型
一般的图	GOTO语句	由指针连接的结构

下面分六节分别介绍四种基本的构造类型、地址类型和指针类型。要使用属于这些类型的数据，必须首先在程序的类型说明部分定义类型本身，再在变量说明部分用已定义的类型说明变量，也可以在说明变量的位置，直接给出类型的定义。并注意，说明类型时，在类型名和它的定义之间用的是等号“=”，在说明变量时，变量名和它的类型之间用的是冒号“:”。

4.1 数 组 类 型

数组类型是一种很常用的构造型数据类型。在标准PASCAL 语言中，它必须由固定数量的同类元素组成。数组类型有两个特点：一方面，它的元素的数量必须是确定的，不允许是变动的；另一方面，它的元素的类型必须是相同的，不允许是混合的。

使用数组类型的最大好处是，可以让一批相同性质的数据共用一个变量名，而不必为每一个数据选定一个名字。不仅程序书写大为简便清晰，提高了程序的可读性，而且便于用循环语句简单地处理这批数据。

在具体介绍数组类型变量的说明与使用之前，先看一个具体例子：计算并输出 0° 至 89° 的正弦函数的数值。实现的程序可以有許多。先介绍用数组方法的，程序如下：

```

PROGRAM ARY00A (OUTPUT);
TYPE
  T=0..89,
  AR=ARRAY[T] OF REAL;
VAR
  I, INTEGER,    R, REAL,
  SINX, AR;
BEGIN
  R:=3.1415926/180,
  FOR I:=0 TO 89 DO
    BEGIN

```

```

    SINX[I]:=SIN (I * R) ;
    WRITE (SINX[I] : 10 : 6) ;
    IF (I+1) MOD 5 = 0 THEN WRITELN
END
END.

```

这个程序非常简单，短短几个语句，就能求出 $0^\circ \sim 89^\circ$ 之间每一度的正弦值，把结果记入数组变量 SINX 的 90 个元素中。

如果不用数组变量，而只用简单变量来表示这 90 个正弦值，那就麻烦得多，上述简短的程序要变成一个很长的程序了。很容易想到，在不用数组的程序中，为了记忆 90 个正弦函数值，就要说明 90 个实数型变量，求 90 个正弦函数值也要用 90 个赋值语句，因为每个正弦值的变量名不同，无法用循环语句处理。

数组类型是由用户定义的。

数组类型定义的格式如下：

——类型名—— = ——ARRAY—— [——下标类型——] —— (OF —— 元素类型——) □

其中：

ARRAY 和 OF 是数组类型的标志，它们都是 PASCAL 语言的保留关键字。类型名由用户自己定义，它必须符合标识符的规定。下标类型一般为简单类型，包括整数类型、字符类型、枚举类型和子界类型。实数类型不行。下标类型一般用子界形式给出。元素类型又叫基类型，它给出的是数组变量元素的类型。它可以是任何数据类型，甚至是构造型数据类型。但是要注意，在同一个数组中，所有元素必须属于同一数据类型。

定义了数组类型之后，就可以用它去说明数组变量了。一般格式如下：

——变量名——： ——数组类型名—— □

在前面的 ARY00A 程序中，在作了类型说明之后作了如下数组类型的变量说明：

VAR

SINX:AR;

表示变量 SINX 属于 AR 数组类型。

数组类型说明和变量说明也可以合并在一起，这样可以减少用户标识符。例如上例中的类型说明和变量说明合并为：

VAR

SINX:ARRAY[0..89] OF REAL;

表达一个数组类型变量项（数组数据元素）的格式如下：

——变量名—— [↑ 表达式] □

其中表达式表示的是数组的下标值。

必须指出，下面的概念要搞清，不能混淆：

数组类型标识符表示的是一种数据类型；

数组类型变量名表示一个数组类型的变量；

SINX[4] 表示一个具体的数组元素的值；

在定义数组类型与使用数组变量时，数组的下标是在方括号“[”和“]”中给出，而不是圆括号“(”和“)”，很多初学PASCAL语言的人，常犯这种用圆括号代替方括号的错误。

在使用数组变量时，PASCAL语言一般不允许把一个整个的数组作为一个单个数据进行访问，每次只允许通过数组变量名与相应下标访问数组的一个元素。只是在两个相同类型的数组变量之间，把一个数组的全体数据赋给另外一个数组时，才可以简单使用数组变量名，而不用给出它们的下标。这种赋值操作，是在每一对相应的下标之间依次进行的。

如果一个数组的元素类型还是数组，那么原来的数组就是二维数组。例如：

VAR ARR: ARRAY[1..5] OF ARRAY['P'..'S'] OF REAL; ARR就是由五个数组（每个又是由四个实数组成的数组）组成的数组。此时为了访问ARR的一个元素数据，要写成ARR[I][CH]的形式。此处I为一整型变量，在这里取值范围为1..5；CH为一字符型变量，在这里取值范围为'P'..'S'。例如，可以用ARR[1]['P']访问该数组的第一个元素数据。

通常情况下，人们更愿意把上述二维数组的说明方便地缩写成：

VAR ARR: ARRAY[1..5, 'P'..'S'] OF REAL; 此时，表示该数组的第i个成分的第j个元素也相应变为ARR[i, CH]的形式。此时可以把ARR看成是由5行4列组成的一个矩阵。

PASCAL语言本身并不限制一个数组下标的个数。下标个数可以为1, 2, 3, ..., n，我们通常称数组下标个数为维，相应的数组就被叫做一维数组，二维数组、三维数组等等。请注意，这里说的是下标个数，而不是某一个下标的取值范围。多维数组所包含的元素个数由每一维的取值个数连乘之积决定。如上面的ARR为二维数组，第一维可取5个值，第二维可取四个值，因此ARR由 5×4 ，即20个元素组成。多维数组的多个下标，其类型可以相同，也允许不同，只要多个下标之间用逗号隔开。

例如，我们按下述方法定义了一种数据类型，又说明了一个数组变量。

TYPE {开始类型说明}

LESSON = (PHYSICS, CHEMICAL, ENGLISH,
MATHEMATICS),

GRADE = 1..5;

AR = ARRAY[1..2000, GRADE, LESSON] OF INTEGER;

VAR {开始变量说明}

SCORE: AR;

它表示AR为三维数组类型，SCORE为三维数组变量。下标1表示学生的学号，取值范围为1到2000；下标2表示学生的年级，取值范围为1到5；下标3表示学生所学的课程，取值范围为物理（PHYSICS），化学（CHEMICAL），英语（ENGLISH）和数学（MATHEMATICS）。

数组SCORE连同相应的下标表示某个年级某名同学某门功课的成绩，三个下标值的顺序、类型与取值范围，必须与类型定义的内容相匹配，否则就要出现错误。

例如，为了表示三年级、学号为500号的同学物理课的成绩，可以写成

SCORE[500, 3, PHYSICS]

由于数组变量SCORE的元素值为整数值，因此可以对它进行赋值、读或写操作。如果数组的元素类型为字符类型，则称这种数组为字符数组。组成这种数组的每一个元素数据都是字符。当计算机系统采用一个字节存放一个字符的方案时，就是字符串。通常字符串被定义为：

```
TYPE STRING=PACKED ARRAY[1..N] OF CHAR;
```

在这里，PACKED是保留关键字，表示每个字符用一个字节存放，N为大于1的正整数。在处理字符数组输入时，多数版本的PASCAL允许在读语句中直接使用数组变量名，当输入字符的个数少于该数组要求时，余下的字符位置用空格代替。标准PASCAL语言没有给出处理字符串的函数，但定义了一个标准的字符串类型ALFA，目前大部分PASCAL语言提供ALFA类型，字符串长度一般为8或10。

ALFA，是PASCAL的标准关键字。

例题：定义两个数组类型，其下标类型分别为子界类型和枚举类型，其元素类型分别为整数类型和实数类型。建立一个包含十个整数元素构成的数组，然后将这数组中的六项内容依次传送给另一个包含六个实数的数组。最后输出这个实数数组的内容。

```
PROGRAM ARY01 (INPUT, OUTPUT);
TYPE
  M=1..10;
  N= (A, B, C, D, E, F);
  P=ARRAY [M] OF INTEGER;
  Q=ARRAY [N] OF REAL;
VAR
  I, M;           J, N;
  X, P;           Y, Q;
BEGIN
  (* PART 1---INPUT ARRAY ELEMENTS *)
  X[2]:=8;
  Y[E]:=3.5;
  FOR I:=1 TO 10 DO READ (X[I]);
  (* PART 2---ARRAY OPERATION *)
  I:=2;
  FOR J:=A TO F DO
    BEGIN
      I:=I+1;
      Y[J]:=X[I]
    END;
  (* PART 3---OUTPUT ARRAY ELEMENTS *)
  I:=0;
  WRITELN ('SIX VALUES OF ARRAY Y IS: ');
  FOR J:=A TO F DO
    BEGIN
```

```

        WRITE (Y[J], ' ', 6) ,
        I:=I+1;
        IF I MOD 2 = 0
        THEN WRITELN
    END
END.

```

这个程序的运行结果可以是：

```

(* INPUT,
11 22 333 4455 6666 16789 32767 110 99 7765
    OUTPUT,
SIX VALUES OF THE ARRAY Y ARE,
3.33000000E+02    4.45500000E+03
6.66600000E+03    1.67890000E+04
3.27670000E+04    1.10000000E+02    *)

```

在这个程序中定义了一个子界类型M和枚举类型N。首先用它们作为两个数组定义中的下标类型，又用它们说明了一个子界型的变量I和一个枚举型变量J，这是为了用循环语句处理数组X和Y而需要的。

再看一个使用字符数组的程序例子。运行结果以注释形式给出在程序清单的下面。

```

PROGRAM ARRY02 (INPUT, OUTPUT) ,
CONST
    C1='PROGRAM FINISHED',
    C2='SIN',
    C3='4567AB',
VAR
    STR1, STR2, STRING (6) ,
    AL, STRING (10) ,
    B, BOOLEAN,
BEGIN READLN,
    READ (STR1, STR2, AL) ,
    WRITELN (STR1, 10, STR2, 10, AL, 12) ,
    B:=STR1=C3,
    WRITELN (B, STR1, C3, C2, C1)
END.
(* INPUT,
ABCDEF123456 @ # $ % ^ & * ( ) -
    OUTPUT,
ABCDEF 123456 @ # $ % ^ & + ( ) -
FALSEABCDEF4567ABSINPROGRAM FINISHED
*)

```

给出这个程序的目的，是为了说明以下几个问题：

(1) 在读入字符(包括字符数组)时,程序开始用一个 READLN 语句,可以避免某些错误结果;

(2) 可以定义与使用字符串常量;

(3) ALFA 可以用来说明标准字符串(MS PASCAL 不支持);

(4) 关于字符串,在相同类型的串常量之间可以进行比较与赋值操作,比较时,是按从前向后的次序,逐个字符进行比较。字符之间的比较,是按它们在ASCII表中排列次序确定大小的,排在前边的比排在后边的小。所有字符都相同的两个串相等。

(5) 字符串输入时,如果输入的字符个数比要求的少,不足的部分自动用空格补足;

(6) 字符串输出时,默认场宽为字符串长度,但也可以由用户指定。

在使用数组时,请注意 PASCAL 语言的有关规定:

数组可以有任意的维数和任意的下标范围,下标只能定义为常量,即没有动态数组;

数组的下标要在方括号中给出,多维数组的多个下标之间用逗号分开,数组的下标值是可以计算得到的,可以用表达式形式给出,读写数组的任何一个元素所用的时间是相同的,与对应的下标值无关。

MS PASCAL 语言还扩充了一个高级数组(SUPER ARRAY)类型。高级数组类型是指只给出数组下标的下界值,而不明确指出数组下标上界值的数组类型。高级数组类型与前面讲过的所有类型都不一样,它不再是单独的一种类型,而代表的是一类类型,或者说是一类类型的一个类型集合。因此,高级数组类型的标识符本身不能用来直接说明变量,多数情况下,是用它派生出相关的类型。此外,还可以用高级数组类型标识符说明过程、函数的变量参数,用在类型子语句中的符号^之后说明指针变量,或用在结构常量中等等。

看下面给出的例子:

TYPE

VECTOR=SUPER ARRAY[1..*] OF REAL;

MATRIX=SUPER ARRAY[1..*, 1..*] OF REAL;

VECT10=VECTOR (10);

MATDEC=MATRIX (100, 100);

VAR

ROW: VECT10;

COL: VECTOR (10);

ROWP: ^VECTOR;

在类型定义部分, VECTOR 和 MATRIX 是两个高级数组的类型标识符。在保留字 ARRAY 前面加上前缀 SUPER、在下标的上界处用 * 代替一个具体的值,定义的就是高级数组类型。

VECT10 和 MATDEC 是另外两个类型标识符,因为它们是用在高级数组类型标识符后,再在括号内给出相应的高级数组类型的上界值的办法定义的,所以,我们说它们是由高级数组类型派生出来的类型。派生出的类型已经是一个具体的类型,它是属于高级数组类型的类型集合中的一种类型。VECTOR (10) 和 MATRIX (100, 100) 被称之为类型派生符,它们的构成规则是在高级数组类型标识符后,再在括号中给出该高级数组的每个

下标的一个确定的上界值。

在变量说明部分，就可以用VECT10和MATDEC来说明变量了。当然，也可以用类型派生符直接说明变量，如对COL的说明那样。

若在过程A中，用VECTOR说明了它的一个变量形参V。在调用过程A时，对应于V的实际参数可以是两种情况：

- (1) 高级数组类型的变量本身（只能出现在过程或函数内部），或指针引用；
- (2) 属于从高级数组类型派生出来的类型的变量或常量。

在PASCAL语言的术语中，把这种用法的参数称之为“同形”数组（Conformant Array）。

UPPER函数可以用来得到高级数组类型的实际参数的下标的上界值。

对ROWP的说明用的是^VECTOR，是把ROWP说明为对高级数组类型VECTOR的指针引用的例子，这允许按所要求的数组大小在程序的堆区（HEAP AREA）分得一片内存区，并通过指针来使用这片内存区的数据。在PASCAL语言的术语中，我们可以称这片内存区为动态数组。这是通过PASCAL语言中的NEW过程来实现的。此时要在NEW过程的实际参数中给出指针变量的同时，还要给出相应高级数组下标的上界值。例如：

```
NEW (ROWP, 10);
```

此时就在程序的堆区建起了由10个实数组成的一个数组。并可以通过ROWP^[1] ROWP^[5]的办法分别访问这个数组的第一个元素、第五个元素等等。

再说明一次，高级数组类型本身不能用来在程序的变量说明部分直接说明变量。高级数组类型只能用来说明过程、函数的变量形式参数（不能用来说明数值形式参数），和说明指针变量，只有高级数组类型的派生符可以用来说明变量、常量或过程、函数的数值参数。

在MS PASCAL语言中，还给出了两种常用的预先定义的高级数组类型，它们就是STRING和LSTRING。其定义为：

```
STRING=SUPER PACKED ARRAY[1..*] OF CHAR;
```

```
LSTRING=SUPER PACKED ARRAY[0..*] OF CHAR;
```

这两种类型，就是我们通常说的字符串类型，是程序设计中很有用的类型。STRING类型除了可以用来说明过程、函数的变量形式参数外，还经常用来在变量说明部分说明指针变量，用它的派生符说明给定长度的字符串变量。对最后一种用法，给出的串长度的最大值为MAXINT-1，即32766。在使用派生符说明变量时，没有默认的长度，用户一定要提供这个长度值。例如：

```
VAR
```

```
    S:STRING (10);
```

```
    S20:STRING (20);
```

不同长度的串变量，属于不同的串类型，既不能彼此赋值，也不能直接比较，这对程序设计是不大方便的。

LSTRING类型，通常被称之为变长字符串类型。它的最大长度为255。LSTRING类型的变量，例如L:LSTRING (10)，其长度为10，它的实际内容在L[1]~L[10]这10个元素中。说它为变长字符串，是指它可以接收长度从1到10的任何一个串变量或串

常量的内容。例如 $L := 'ABC'$ 语句将把 'A'、'B'、'C'、三个字符分别放到 L 串变量的前三个元素中，并记下此时串的实际长度（有效长度）为 3，而 L 的后七个元素的内容是不能使用的。对 LSTRING 类型的变量，要把它的长度保存在变量的第 0 号元素中。在执行过上面一个赋值语句后， $L[0]$ 的内容为 $CHR(3)$ 。从上面给出的 STRING 和 LSTRING 的定义可以看到，LSTRING 的下标下界值为 0，这个 0 号元素就是专门用来记忆 LSTRING 的实际长度的。这个长度必须以字符形式给出，而且编码值不能大于 255。

不同长度的 LSTRING 类型的变量可以直接比较大小，也可以把比较短的串变量的值赋给长的串变量。这正是 LSTRING 类型比 STRING 类型用得更多、使用更方便的原因所在。通过检查 LSTRING 变量的 0 单元的内容，就可以知道这个串变量的实际长度。MS PASCAL 还提供了一个检查变长字符串实际长度的函数。具体用法给出如下：

```
VAR
    L: LSTRING (10);
    Y: INTEGER;
    X: BYTE;
BEGIN
    L := 'A1B2C3';
    Y := ORD (L[0]);
    X := L.LEN;
    :
```

这里 Y 和 X 都保留着变长串变量 L 此时的实际长度，其值均应为 6。从例子中可以看到，可以用访问 L 的 0 号单元 $L[0]$ 并变字符为相应 ASCII 值的办法得到 L 串的实际长度，也可以用在串名后用一个，'.LEN' 的形式，直接得到 LSTRING 变量的实际长度，此长度是用 BYTE 类型表示的，这是 MS PASCAL 提供的得到 LSTRING 变量实际长度的扩展功能。

请注意，若把 $L := 'A1B2C3'$ 变成 $L := '12345678'$ ，则不能简单地如下实现：

```
for CH:='1' TO '8' do L[ORD(CH)-48]:=CH;
```

这是因为用这种办法不能改变 L 变量的实际长度。若用 $WRITELN(L)$ 输出 L 的内容，将显示出 123456，长度 6 是上一次执行 $L := 'A1B2C3'$ 时得到的。正确的作法是，在 for 语句之后，再用一个 $L[0] := CHR(8)$ 语句变更 L 串的实际长度，这里的长度 8 要用 CHR 函数将其转换成字符编码，以字符变量形式写入 $L[0]$ ，不能写成 $L[0] := 8$ ，因为 $L[0]$ 和整型量 8 类型不一致。同样，在上面的 for 语句中，把字符 '1' 写进 $L[1]$ 中，下标值 1 也是通过求得字符 '1' 的 ASCII 编码值后再减去 $ORD('0')$ ，即减去 48 得出来的。

必须说明，虽然 LSTRING 的定义为 $SUPER\ PAKED\ ARRAY[0..*]\ OF\ CHAR$ ，但 LSTRING (10) 与用户在自己程序中用 $PAKED\ ARRAY(0..10)\ OF\ CHAR$ 定义的变量不属于同一类型。原因很简单，LSTRING (10) 变量的 0 号元素是用于记该串变量的实际长度的，而后者的 0 号元素与它的 1 号到 10 号元素都是用于记忆字符的，因此可以记忆 11 个字符。而且用这种办法说明的变量与 STRING (11) 也不属于同一类型。因为 STRING (11) 的下标的下界值为 1，而不是 0。

为了更方便地使用 STRING 和 LSTRING 类型的变量，MS PASCAL 提供了许多过

程和函数，有关内容请参见第五章的有关章节。

4.2 记录类型和 WITH (开域) 语句

4.2.1 记录类型

有记录类型数据是 PASCAL 语言的一个重要优点。记录类型是一种比较复杂但非常灵活的构造型数据类型。它是由固定数量、类型却可以不相同的元素（又称为域）组成。元素的数量必须固定，这与数组类型相同。但是，元素的类型允许不同，这是与组数类型的一个重要区别。正是由于这一点，决定了访问它的元素的方法与访问数组元素完全不同。

例如，为了定义一个表示由年、月、日三项组成的日期的数据类型 DATE，可以通过如下方法进行类型定义，

```
TYPE
```

```
    DATE=RECORD
```

```
        YEAR:INTEGER;
```

```
        MONTH: (JAN, FEB, MAR, APR, MAY, JUN,  
                JUL, AUG, SEP, OCT, NOV, DEC);
```

```
        DAY:1..31
```

```
    END;
```

这里的 RECORD 和 END 是定义记录类型的保留关键字。

其中，

DATE 类型由三个元素组成：YEAR，MONTH 和 DAY，

这三个元素分别属于三种数据类型，

YEAR 属于整数类型，

MONTH 属于枚举类型。括号中的标识符是英文一月至十二月的缩写，

DAY 属于子界类型。它表示可以 1 日到 31 日

如果有两个变量 X 和 Y，它们都属于 DATE 类型，则可以说明如下，

```
VAR
```

```
    X, Y:DATE;
```

例如，用 X 表示 1949 年 10 月 1 日，用 Y 表示 1986 年 7 月 1 日，可以通过如下赋值语句来实现，

```
X • YEAR := 1949;
```

```
Y • YEAR := 1986;
```

```
X • MONTH := OCT;
```

```
Y • MONTH := JUL;
```

```
X • DAY := 1;
```

```
Y • DAY := 1;
```

从这些赋值语句可以看出，表示一个记录元素与表示一个数组元素的方法完全不同。数组元素是通过变量名与相应的下标来表示的，而记录元素是通过变量名与域名来表示。其中变量名与域名之间用一个圆点相连接。

关于记录类型说明、域表、记录类型变量说明和记录元素的格式如下图所示，

——类型名—— = ——RECORD
域表
END——□

(A) 记录类型说明的一般格式

——↑—— 域名—— : ——类型—— □
↑
.....
.....

(B) 域表的格式

——↑—— 变量名—— : ——记录类型—— □
↑
.....

(C) 记录类型变量说明的格式

——变量名——↑—— • ——域名—— □

(D) 记录元素的格式

将记录类型定义和变量说明合并起来是允许的。但是，由于记录类型比较复杂，单独说明可以更加清楚一些。

域名和域的类型的关系，与变量名及其对应类型之间的关系相同。允许几个域名属于同一种类型，但不允许一个域名同属于几种类型。

一个域可以属于任何类型，甚至是一个记录类型。例如记录学生情况的记录类型 STUDENT 中，成绩域 SCORE 本身也可以是一个记录，登录政治、数学、物理、英语及其平均分。

例如：

TYPE

STUDENT = RECORD

NO: INTEGER,

NAME: PACKED ARRAY [1..15] OF CHAR;

SEX: (MALE, FEMALE);

AGE: 18..25;

SCORE: RECORD

POLITICS, MATHEMATICS,

PHYSICS, ENGLISH: INTEGER;

AVERAGE: REAL

END

END;

VA

S STUDENT;

在这种情况下，表示某名同学的某门功课的成绩时，必须用二个域名：一个是域名 SCORE，一个是小域名，如 ENGLISH。比如某学生英语成绩是 85 分，就要用下面赋值语句：

```
S • SCORE • ENGLISH := 85;
```

使用记录型数据时，还有一个好处，允许在它的最后那个域里登录相同性质、不同结构与内容的数据。此时，这个域被称为记录的变体 (Variant) 部分。这是很有用的。例如，我们希望反映学生的政治情况，即是否党、团员？何时入党（团）？是否转正？每个学生的情况不同，这一栏要填的内容就不同。这里存在一个选择问题。在 PASCAL 语言中，解决这类问题必须用记录的变体部分。

下面是学生情况的新的记录类型定义：

```
TYPE
    POLITICS = (CP, CY, MASS);
    STUDENT = RECORD
        NAME: ARRAY[1..15] OF CHAR;
        AGE: 18..25;
        CASE STATUS: POLITICS OF
            CP: (INCP: INTEGER,
                STATE: BOOLEAN);
            CY: (INCY: INTEGER);
            MASS: ( )
        END;
```

在类型 STUDENT 的定义中，可以划分成两部分。CASE 之前的，是这个记录型的固定部分，与我们前面讲的记录定义相符合。CASE 之后给出的就是该记录型的变体部分。

定义记录型变体的一般格式表示如下：

```
——CASE——标志域名——:——标志域类型——OF——
|
|
|
↑↑情况标号——:——(——域表——)——END——
```

这里

CASE、OF 和 END 是 PASCAL 的保留关键字。

标志域名是用户为变体域选的域名，要符合标识符定义规则。在它之后要给出标志域类型，通常应为枚举型，此时也可省略掉标志域名，直接用类型标识符代替。

情况标号为常量，应与标志域类型吻合，用于选择变体域中的变体成分。

这里的域表与记录类型定义中的域表结构和功能完全相同，给出相应变体成分的数据组成方式和类型。

回过头去看一下刚刚定义的类型 STUDENT。在那里，标志域名为 STATUS，属于反映政治面貌的 POLITICS 类型，假定 CP 代表党员，CY 代表团员，MASS 代表非党团员群众。反映这三种情况的三个变体成分的内容是不一样的。党员，要登记入党年份和是否转

正两个数据：团员，则只需登记入团年份即可；一般群众，不登任何数据。属于这种类型的数据，在变体部分只能从三个变体成分中选取其一。显而易见，反映一名党员学生和一名群众的两个数据，在变体部分就很不一样，但这两个数据属于同一类型。

变体成分，在一个记录中只能有一个，并且只能是记录的最后一个域。变体的域表里还可以有变体成分，即变体成分可以嵌套。

当变体内多个变体成分占用的存储空间要求不同时，按最多者分配。使用变体成分不是为了、也不能节省存储空间。一个记录既可以只有固定部分，也可以只有变体部分，或两个部分都有。

4.2.2 WITH (开域)语句

开域语句是为更方便地使用记录型数据而设置的，其格式如下：

——WITH——↑——记录名——DO————语句——□

其中：

WITH和DO为开域语句的标志，是PASCAL语言的保留关键字；语句指开域语句的成分语句；记录名为记录型变量名或记录型域名。

“开域”就是打开疆域，它有效地指定记录变量名或记录型域名的管辖范围，使在它们名下说明的域名可以象变量那样灵活地使用，这样书写或阅读程序都更为方便。

例如，前面定义了记录类型DATE，又说明了DATE类型的变量X和Y，要表示1949年10月1日，可以用下述语句：

```
WITH X DO
```

```
  BEGIN YEAR:=1949; MONTH:=OCT; DAY:=1  END;
```

这样，在X变量的管辖范围内（WITH的子语句中），在X的每个域名之前就不必都冠以变量名X了。使程序书写简便，清晰，阅读起来也更方便。

当一个记录的某个域也为记录型时，如前面讲到的登记学生四门课程的成绩的记录。可以把 S·SCORE·ENGLISH:=85 表示为：

```
  WITH S DO
```

```
    WITH SCORE DO
```

```
      ENGLISH:=85;
```

或表示为：

```
  WITH S, SCORE DO ENGLISH:=85;
```

两种表示方法的功能完全一样。第一种方法更直观地表现了WITH语句的嵌套用法。

使用WITH语句的一条限制是，不允许在WITH的子语句内部改变开域变量。这种错误大多发生在以记录为数组的元素类型的情况下。例如：

```
  WITH A[1] DO
```

```
    BEGIN
```

```
      :
```

```

        I:=I+1;
        :
    END;

```

I的改变，也就改变了A[I]，程序不能正常运行。请注意，这里的A[I]是一个记录类型数组的一个记录型数据成分。

在以后讲到指针型变量时，它用WITH语句中的这种错误几率更高。

一般地说，WITH语句嵌套的层数不应太多，例如不大于四。

下面给出一个应用记录型数据的程序。这个程序用来读入四名学生的情况到记录型数组中，包括学号、姓名、性别和三门课程的考试成绩，并求出平均成绩。这个程序中需要说明的一点是对性别域的处理办法。性别域被定义为枚举类型，标准PASCAL语言规定不能直接用READ语句读入它的值，而是用读入一个字符后检查字符是否为'M'和'F'并向性别域赋以MALE或FEMALE的值的值的方法实现的。这个程序中没有用WITH语句，因此显得不精炼。

在这个程序中，还定义了一个枚举型变量LSN，用它作为一个FOR语句的循环控制变量，读入学生各门课程的考试成绩。这是在处理下标类型为枚举型的数组时经常采用的办法，程序显得精炼清楚。这个程序只是一般地表明记录型数据在程序中的使用方法。

```

PROGRAM STUDENTS (INPUT, OUTPUT);
TYPE
    LESSON= (MAT, PHYS, ENGL);
    ST=RECORD
        NUMBER; INTEGER;
        NAME; PACKED ARRAY[1..10] OF CHAR;
        SEX; (MALE, FEMALE);
        SCORE; RECORD
            SCR; ARRAY[LESSON] OF INTEGER;
            AVERAGE; REAL
        END
    END;
VAR
    STUDENT; ARRAY[1..4] OF ST;
    I, J; INTEGER;
    CH; CHAR;
    LSN; LESSON;
    AL; PACKED ARRAY[1..6] OF CHAR;
BEGIN
    (* PART1: READ STUDENT'S INFORMATION *)
    FOR I:=1 TO 4 DO
        BEGIN
            WRITELN ('STUDENT', I:2);
            WRITE ('NUMBER= ');

```

```

        READLN (STUDENT[I].NUMBER);
        WRITE ('NAME=');
        READLN (STUDENT[I].NAME);
        WRITE ('SEX—MALE (M) OR FEMALE (F) :');
REPEAT
    READLN (CH);
    IF CH='M'
        THEN STUDENT[I].SEX:=MALE
    ELSE
        IF CH='F'
            THEN STUDENT[I].SEX:=FEMALE
        ELSE WRITELN ('ANSWER NO CORRECT')
UNTIL (CH='M') OR (CH='F');
    J:=0;
    WRITE ('MAT, PHS, ENGL SCORE:');
    FOR LSN:=MAT TO ENGL DO
        BEGIN
            READ (STUDENT[I].SCORE.SCR[LSN]);
            J:=J+STUDENT[I].SCORE.SCR[LSN];
        END;
    STUDENT[I].SCORE.AVERAGE:=J/3
END;
    (*PART2:OUTPUT STUDENT'S INFORMATION*)
FOR I:=1 TO 4 DO
    WITH STUDENT [I] DO
        WITH SCORE DO
            BEGIN
                IF SEX=MALE
                    THEN AL:='MALE'
                ELSE AL:='FEMALE';
                WRITE (NUMBER:4, NAME:14, AL:8);
                FOR LSN:=MAT TO ENGL DO
                    WRITE (SCR[LSN]:4);
                WRITELN (AVERAGE:7:1)
            END
        END;
END.

(*
STUDENT 1
NUMBER=1
NAME=L! PENG

```

```

SEX=MALE (M) OR FEMALE (F) ;F
MAT, PHS, ENGL SCORE;85 90 98
STUDENT 2
NUMBER= 2
NAME=WANG DALI
SEX=MALE (M) OR FEMALE (F) ;M
MAT, PHS, ENGL SCORE;85 100 92
STUDENT 3
NUMBER= 3
NAME=GAO XIN
SEX=MALE (M) OR FEMALE (F) ;M
MAT, PHS, ENGL SCORE;100 62 80
STUDENT 4
NUMBER= 4
NAME=ZHAO DAYAN
SEX=MALE (M) OR FEMALE (F) ;F
MAT,PHS,ENGL SCORE;80 80 80

```

1	LI PENG	FEMALE	85	90	98	91.0
2	WANG DALI	MALE	85	100	92	92.3
3	GAO XIN	MALE	100	62	80	80.7
4	ZHAO DAYAN	FEMALE	80	80	80	86.0

*)

4.3 集合类型

如果把某些具有共同特性（共性）而又能互相区别（个性）的对象聚集在一起，这样形成的整体就叫集合（SET）。一个集合是同类型对象的一个汇集。集合中既互相联系又互相区别的对象，叫这种集合的元素。没有任何元素的集合是空集合。

例如，所有的大写英文字母是一个集合，它包括26个元素，A B C……Z。其共性在于都是英文字母，其个性在于各自形状不同，读音不同。

在 PASCAL 语言中，集合类型定义为一个某种数据类型的值的集合，这个集合是它的元素类型的势集（POWERSET）。也就是说，集合类型定义为类型的值的所有子集（包括空集在内）的集合。其中元素类型必须是简单类型，包括枚举类型与子界类型。

集合是 PASCAL 语言中的构造类型数据之一。它与其它构造型数据的共同之点是，它们都是收集起来的各自的一批属于基类型数据，但差别是明显的。人们总是把一个集合变量看作为一个完整单一的数据使用，绝对不允许去访问它的任何一个元素。而其它三种构造类型数据恰好相反，通常是把它们的一个元素作为一个独立的数据使用。

集合类型定义的格式如下：

——类型名—— = ——SET——OF——元素类型——□

其中;

SET 和 OF 是集合类型的标志, 都是保留关键字;

类型名由用户自己定义, 必须符合标识符的有关规定;

元素类型必须是某些简单类型, 包括枚举类型和子界类型。

例如, 我们可以定义三个集合类型如下:

```
TYPE
```

```
  A = 1..20;
```

```
  M = (A,B,C,D,E123,F456);
```

```
  SETA = SET OF A;
```

```
  SETB = SET OF M;
```

```
  SETC = SET OF CHAR;
```

这之后 我们就可以用它们去说明集合型变量了。和其它类型变量说明一样, 在说明集合型变量时, 也可以把类型定义与变量说明合在一起进行, 如:

```
VAR
```

```
  S: SET OF 1..20;
```

为了高效率地进行集合运算, PASCAL 语言的报告中, 建议限制集合类型中元素的个数。MS PASCAL 语言版本规定, 集合类型中元素的个数不能超过256个。因此, 如果一个集合类型的元素类型是字符类型, 那么, 这种集合类型将可以包括编码从0到255的全部256个字符。标准 PASCAL 语言和其它版本的 PASCAL 语言, 允许的集合元素往往比256要小, 例如规定为128, 甚至于64。此时, 同样给出:

```
SETC = SET OF CHAR;
```

所包括的真实内容就很不一样。对MS PASCAL语言, 它相当于:

```
SETC = SET OF CHR(0) .. CHR(255);
```

对规定最多只有64个元素的其它PASCAL语言, 它可能相当于:

```
SETC = SET OF ' ' .. '~' ;
```

即编码值从32开始的空格到95的连字符。

使用集合类型时, 一定要先阅读所用PASCAL语言的用户手册, 了解对集合最多元素个数的具体规定。

如果在说明或使用集合类型时, 其元素个数超过了语言规定的最多允许的值, 那么, 在进行编译时会指出“集合元素有错”(BAD SET ELEMENT)。例如语句: $Z := ['\sqcup' .. '\sim']$, 对最多只允许64元素的字符型集合就是非法的, 其中字符编码大于95的字符不在这个集合类型之中。

对集合变量可以进行赋值运算, 还可以进行关系运算而得到布尔型结果。关系运算只能在相同类型的集合变量间进行。

集合类型数据进行关系运算的格式如下:

——集合1——关系运算符——集合2——□

集合关系运算符一般有四种, 即:

= 表示两个集合相等, 即两个集合中的元素完全相同。

<> 表示两个集合不等, 即两个集合中的元素不完全相同。

⊃= 表示前者蕴含后者, 即后者中的元素都能在前者中找到。

\leq 表示前者蕴含于后者，即前者中的元素都能在后者中找到。

请注意，关系运算符“ $>$ ”和“ $<$ ”不能运用于集合的比较。

集合不能进行与、或、求反等逻辑运算，因为集合类型数据不是布尔值。但是，它们可以进行四种特殊的逻辑运算，通常叫集合运算。集合变量的元素必须在方括号中给出。

(一) 属于 (IN) 运算

IN用来判断某些元素是否已在集合中了，运算结果为布尔型。IN的使用可以大大提高集合的使用价值，它的处理速度比用其它办法快得多。

如 $A \text{ IN } [A, B, C]$ 为真

$B \text{ IN } [A, C, D, E]$ 为假

(二) 并 (UNION) 运算

设M、N为两个集合。由属于M或者属于N的全部元素，即M集合中的元素，加上N集合中的与M集合中不同的那些元素组成的新集合，称为集合M和N的并。并运算符为“ $+$ ”，记作 $M+N$ 。

例如：

$[A, B, C] + [D]$ 为 $[A, B, C, D]$ ；

$[A, B, C] + [B]$ 为 $[A, B, C]$ ；

$[A, B, C] + [B, C, D]$ 为 $[A, B, C, D]$ ；

(三) 交 (INTERSECTION) 运算

设M、N为两个集合。由既属于M又属于N的所有元素，即两个集合中都拥有的那些元素组成的新集合，称为集合M和N的交。交运算符为“ $*$ ”，记作 $M * N$ 。例如：

$[A, B, C] * [B]$ 为 $[B]$ ；

$[A, B, C] * [B, C, D]$ 为 $[B, C]$ ；

$[A, B, C] * [D]$ 为 $[\]$ ；

(四) 差 (DIFFERENCE) 运算

设M、N为两个集合。由属于M而不属于N的所有元素，即M集合中的元素除去N集合中那些与M集合中相同的元素后，由剩下的所有元素组成的新集合，称为集合M和N的差。差运算符为“ $-$ ”，记作 $M-N$ 。

例如： $[A, B, C] - [B]$ 为 $[A, C]$ ；

$[A, B, C] - [D]$ 为 $[A, B, C]$ ；

$[A, B, C] - [B, C, D]$ 为 $[A]$ ；

由于这三种集合运算的存在，赋值语句的应用又可以进一步扩展。赋值语句的右边可以是集合表达式。例如：

$I := 5$ ；

$Y := [1, 2, 3, 4] + [I]$ ；

运行之后，集合Y中就有五个元素：1，2，3，4和5。

但是如下语句：

$Y := [1, 2, 3, 4] + I$ ；

显然是错误的。因为I是整数类型而不是集合型变量，如果要变成集合中的元素，必须加上方括号 $[\]$ 。

集合运算必须在相同类型的集合间进行。

不能对集合进行算术运算，也不能直接用输入输出语句对集合变量进行操作。

集合变量表示的是一个整体数据，只能经过它的变量名直接使用。这与其它三种构造型数据的用法截然不同。

在使用集合处理某些问题时，可以大大节省内存空间并大大提高解题速度。因此，在处理这样的问题时，要尽可能地使用集合型数据。

下面给出两个使用集合的程序例子。第一个程序用于求出 2 到 255 之间的全部素数。素数又叫质数，是只能被 1 和它本身整除的正整数，如 5 和 7 就是素数，而 8 和 9 都不是素数。

这里在程序中用的是筛选法。首先把 $2 \cdots 255$ 放入筛集合 S 中。接下来把筛集合中的最小值取来作为找到的一个素数放入素数集合 P 中，并把筛中这个素数以及它的全部整数倍的值筛掉。然后再找出剩在筛中的最小值，继续重复上面的操作，直到筛集合 S 变为空。最后一步，是按集合 P 中的内容，输出运算结果。必须再次强调指出，在 PASCAL 语言中是不允许用 WRITE 语句直接输出任何一个集合变量的内容的。我们这里用的是判别某个值是否属于集合 P，是则输出该整型值，否则接着往下判别。

```
PROGRAM SETS (OUTPUT);
CONST N=255;
TYPE NUMBSET=SET OF 2..N;
VAR S, P, NUMBSET;
    I, J, K, L, INTEGER;
BEGIN
    (* INITIAL VALUES *)
    S:= [ 2..N ];
    P:= [ ];      J:= 2;
    (* GET ALL PRIMER NUMBERS *)
    REPEAT
        IF J<=N THEN
            BEGIN
                L:=J;      P:=P+ [ J ];
                END;
            WHILE L<=N DO
                BEGIN
                    S:=S-[ L ];      L:=L+J;
                END;
            WHILE (J<=N) AND NOT (J IN S) DO J:=SUCC (J)
        UNTIL S=[ ];
    (* OUTPUT THE RESULT AT *)
    K:= 0;
    WRITELN ('ALL PRIMES IN 2..255 ARE: ');
    FOR I:=2 TO N DO
        IF I IN P THEN
```

```

        BEGIN
            WRITE (1:5) ;      K:=K+1;
            IF K MOD 5 = 0 THEN WRITELN
        END;
    WRITELN
END.

```

(*

ALL PRIMES IN 2..255 ARE:

2	3	5	7	11
13	17	19	23	29
31	37	41	43	47
53	59	61	67	71
73	79	83	89	97
101	103	107	109	113
127	131	137	139	149
151	157	163	167	173
179	181	191	193	197
199	211	223	227	229
233	239	241	251	

*)

第二个程序用来把从终端输入的一行字符中满足标识符和整型、实型数字定义的内容找出来并在终端上显示。标识符的定义为英文字母开头的可由英文字母和数字字符组成的一个字符序列。整数为数字字符 0 ~ 9 组成的数字字符序列，实数是数字字符序列中还可以有个小数点。

```

PROGRAM SETS1 (INPUT, OUTPUT) ;
VAR
    I: INTEGER;
    AR: PACKED ARRAY[1..78] OF CHAR;
    (*   AR1: LSTRING (78)   *)
    LETTERS: SET OF 'A'..'Z';
    DIGITS: SET OF '0'..'9';
BEGIN
    LETTERS := [ 'A'..'Z' ];
    DIGITS := [ '0'..'9' ];
    FOR I:=1 TO 78 DO AR[I] := ' ';
    (*   AR1:=NUL   *)
    READLN (AR);      I:=1;
    REPEAT
        WHILE (AR[I] = ' ') AND (I<78) DO I:=I+1;
        IF AR[I] IN LETTERS

```



```

    THEN
        BEGIN
            WHILE((AR[I] IN LETTERS) OR (AR[I] IN DIGITS))
                AND (I<78) DO
                    BEGIN
                        WRITE (AR[I]);    I:=I+1
                    END;
                WRITELN
            END ELSE
IF AR[I] IN DIGITS
    THEN
        BEGIN
            WHILE (AR[I] IN DIGITS) OR (AR[I]='.') DO
                BEGIN
                    WRITE (AR[I]); I:=I+1
                END;
            WRITELN
        END
    ELSE I:=I+1
UNTIL I> 78
END.

```

```

( *
123 ASD 456, D 2340495" , 123><? DJFKJDK ? 12 DF5KJD 1.23,
123.456,
.23
ASD456
D2340495
123
DJFKJDK
12
DF5KJD1
23
123.456
*)

```

这个程序中用了按一定结构组织起来的多种语句，但不难看懂。

思考题：

这个程序中为集合变量赋初值的语句可以省略吗？为什么？

这个程序在输出实数时可能遇到什么样的错误？请您把它修改正确。

这个程序中的 READLN (AR) 和 FOR I:=1 TO 78 DO READ (AR[I]) 在功能上完全相同吗？请说明理由。

读者认真思考以上问题后，对有关内容的掌握会是很有帮助的。

4.4 文件类型

4.4.1 有关文件的基本概念

文件属于构造类型的数据。计算机系统文件，通常指的是存放在外存储器（如磁盘、磁带等）上的一批信息的集合。广义地讲，某些慢速的输入输出设备（如行式打印机、终端等）也被看作为文件。

每个文件都有自己的名字，通称文件名。用户须通过指定文件名来读写自己的文件内容。标准PASCAL语言处理文件的功能不很完备，它只处理顺序文件。

顺序文件，是指由一批相同类型的数据组成的一个数据序列。每一个数据，是文件的一个组成成分，通称文件的一个记录，它可以是文件类型之外的其它任何类型的数据。注意，要把文件记录与前面讲的记录类型的概念严格区别开来，这二者指的不是同一个概念。对顺序文件来说，各记录之间的排列次序是有实际意义的，是顺序关系。用户也只能按记录的实际排列次序，依次访问（读或写）文件的每一个记录。在任何一个时刻，文件中只有一个记录的内容是能直接使用的。这些是对使用顺序文件所加的重要限制。必须指出，不少PASCAL编译程序都扩充了一些功能，也允许用户使用其它类型的文件。用户可以参阅有关手册，我们这里不叙述这些扩充功能。MS PASCAL就还允许用户使用直接文件。

我们通常把一个文件所含有的元素个数称为文件长度。如果一个文件的长度为零，就是说，它不包含任何元素，则说这是一个空文件。文件的长度，不是通过文件说明来确定的，它可以随着对文件处理的进展情况，随时加以改变。在使用顺序文件时，要在文件中间位置做增加或删除一个记录的事，是很困难的。

在PASCAL语言的各种类型的数据中，文件类型的数据用法最复杂。对此，可以简单而形象地解释如下。用户程序中的其它类型的数据，是用户程序本身的一个组成成分，总是和用户程序在一起，程序运行时，随时可以直接地使用它们，就象装在某人自己书包中的书，他随时可以取来翻阅一样。而对文件数据执行的最基本的操作，是从已有文件那里把数据读到程序中来，或把程序中产生的数据写到一个文件中。这些都涉及到使用计算机的外部设备，例如使用什么类型的、哪一台具体设备，要进行什么操作，怎么样让用到的那台外设与用户的程序协同动作等等。应该说，计算机系统文件，包括外设，可不是用户程序本身的一部分。程序中使用它们，但得经过“提出申请”，“办理手续”，“说明要求”等步骤。这与我们要使用图书馆中的藏书有许多类似之处。

在PASCAL程序中，“提出申请”由文件说明解决。“办理手续”和“说明要求”由语言所提供的几个标准过程和标准函数来完成。会用这些标准过程和标准函数是使用文件的关键。人们也称这些对文件进行操作的标准过程为文件处理的操作符。

我们可以从不同的角度来看待MS PASCAL语言使用的文件。

从文件的结构看，习惯上把PASCAL语言的文件区分为二进制（BINARY）文件和ASCII文件（TEXT）两大类。前者又被称为无格式文件，只能用GET和PUT两个标准过程对它执行读写操作，每次传送的一个记录的长度，就是由定义该二进制文件时规定的文

件记录类型要占用的字节数。后者也常被称为字符文件或文本文件、正文文件。它是由字符组成的文件，属于有结构的文件，即若干个字符被分成为一行，若干行又可以被分成一页。每行都用行结束符 (LINE MARKER) 来标记行的结束，通常是用 CHR (13)，即回车 (Carriage Return) 键的编码作为行结束符。从概念上讲，行结束符本身不是字符文件的一个有效字符，它只起到格式控制作用，因此，用 READ 语句读字符文件时，这个行结束符将被读为空字符 CHR (0)，在某些场合，也可能被读为空格 (CHR (32))。此外，页结束通常用 CHR (12) 表示，文件结束用 CHR (26)，即 CTRL/Z 键的编码值表示。它们也属于控制字符。对字符文件，既可以用 GET 和 PUT 两个文件操作符执行读写 (认为字符文件是二进制文件的一个特例)，也可以用 READ 和 WRITE 两个语句执行读写。

从文件的读写方式看，又可以把 PASCAL 的文件区分为终端 (TERMINAL) 文件、顺序 (SEQUENTIAL) 文件和直接 (DIRECT) 文件三大类。

终端文件总是对应着交互入出的终端设备或打印机设备。

顺序文件对应的是磁盘上的文件或其它顺序访问的设备。

直接文件对应的是磁盘上的文件或其它随机访问的设备。

请注意，文件的结构和访问方式，是从两个不同的角度来表示文件的特性的。结构反映的是文件的组成规则，即文件记录是只能由字符组成 (字符文件，文本文件) 呢，还是也可以由二进制表示的不同类型的数据组成 (二进制文件)。访问方式反映的是读写文件的方式方法，即是严格地按记录的顺序依次读写呢，还是按记录号选择读写。对于终端和打印机，是对每一个字符直接入出呢，还是按行入出。这二者之间有某些联系，但反映的不是同一个方面的问题。

如果用户不做特殊说明，系统就默认用户的文件为 BINARY 结构和 SEQUENTIAL 方式。

对于预先说明的两个标准文件 INPUT 和 OUTPUT，系统总是认为它们代表终端的键盘和终端的显示器，并默认它们为 ASCII 结构和 TERMINAL 方式。此时，入出操作都是以行为单位执行的，就是说，用户输入的每一个字符将首先被输入到 DOS 操作系统设置的一个缓冲区中，并同时显示到屏幕上。但用户的程序只能在用户打入了回车键之后，才能真正接收到已输入了的一行中的完整内容。用户程序是不能在每打入一个字符之后就立刻读到它。用户也可以重新指定 INPUT 和 OUTPUT 两个文件的结构，也可以给出其它工作方式。

为了让自己的程序能随时读到从键盘打入的每一个字符，可以指定终端为 BINARY 结构的 TERMINAL 方式工作的文件，就是把终端说明为 FILE OF CHAR。这样，用户就可以在自己的程序中用 GET 过程随时读到从键盘打入的每一个字符。这种方法，在实现屏幕编辑程序，实现功能清单选择 (MENU SELECTION) 和其它一些不能以行为单位处理键盘输入的应用领域的程序设计中，是非常有用的。此时，不能用 READ 过程访问终端，只能用 GET 过程。并且，有两个字符，CHR (0) 和 CHR (255) 不能正常使用。CHR (0) 被用来表示尚无数据可读，即执行到 GET 过程时，用户尚未敲键盘，程序要等待。CHR (255) 不能被写入到 TERMINAL/BINARY 文件中，因为它代表读一个字符到 DOS 操作系统。程序例子在后面给出。

直接文件 是通过在打开文件之前，设置相应文件的 MODE 域为 DIRECT 来指明的。

在直接文件上,可以按给定的记录号执行读写。与顺序文件相比,操作直接文件的优点是:

- 可以按记录号读写文件,而不再是按记录的严格次序依次读写。
- 可以对同一个文件执行既读又写两种操作,写的时候,是按指定的记录号写一个记录到文件中,而不再是总写在文件的最末尾处。

4.4.2 文件的说明

标准PASCAL语言规定,要在程序中使用文件,必须在程序的适当部分对它加以说明。通常情况下,这种说明可以出现在如下两个地方:

(1) 应在程序首部的文件参数表中给出该程序用到的全部文件的文件名。例如,

```
PROGRAM ABC (INPUT, OUTPUT, A, B, C);
```

这表明在这个叫ABC的程序中,用到了五个文件,它们的名称分别叫做INPUT, OUTPUT, A, B, 和C。

(2) 应在程序的说明部分中,说明这些文件的类型,说得更准确些,是说明组成每个文件的成分的类型。例如,

```
VAR
    INPUT, OUTPUT: TEXT;
    A: FILE OF INTEGER;
    B: FILE OF REAL;
    C: FILE OF RECORD
        LINK: INTEGER;
        NAME: LSTRING (15);
        AGE: 0..100
END;
```

在变量说明部分,对文件说明的格式为:

——标识符——:——FILE——OF——类型——□

这里的标识符是文件名,它应符合PASCAL语言对标识符的有关规定。

后面的冒号, FILE与OF是必须按此格式给出的。正是这里的FILE这个关键字,表明说明的是一个文件。

最后的类型,既可以是一个类型标识符(标准的,或已定义过的),也可以是一个新的类型定义。当然,我们也完全可以先在类型说明部分定义一个文件类型标识符,再用它说明文件变量。但这里的类型必须不是文件类型, file of file……是不合法的。

在这里,要解释几个概念:

(1) INPUT, OUTPUT是标准文件名,通常是指计算机系统中默认的输入输出名的字符型设备,因此无需在VAR部分再对它说明。但在程序首部中不能省略不写;

(2) TEXT是标准类型标识符,可以直接使用,表示字符文件,或称正文文件、文本文件;

(3) 在文件说明中给出的文件名是在程序的读写语句中使用的,通称内部文件名,不是磁盘上实际文件用的文件名(可称为外部文件名);怎样指出某一个内部文件与哪个外部的实际文件相对应,我们将在下一小节说明;

(4) 在VAR部分说明文件, 意味着已定义文件为变量, 变量名就是内部文件名。我们使用文件数据, 即读写文件, 每次都以文件的一个记录为单位进行。正在处理的那个记录被称为文件的“窗口”, 用在文件名后加一个字符“↑”或“^”表示, 如 A↑, B↑, C↑等等, 它们又被称为文件变量缓冲区。所以, 说明了一个文件后, 在用户的程序中, 自动引入了这个文件的一个变量缓冲区, 其类型就是文件的成分类型, 而不是整个文件。在每一时刻, 只有在这个缓冲区中的那个文件数据是可以直接使用的。

MS PASCAL规定, 如果文件名出现在程序参数表中 (INPUT与OUTPUT两个文件名例外), 程序运行时, 系统将提示用户指定该文件名对应的实际文件名。如果用户希望取消此项操作, 就不要把有关文件名写进程序参数表中, 只在变量说明部分说明这样的文件。

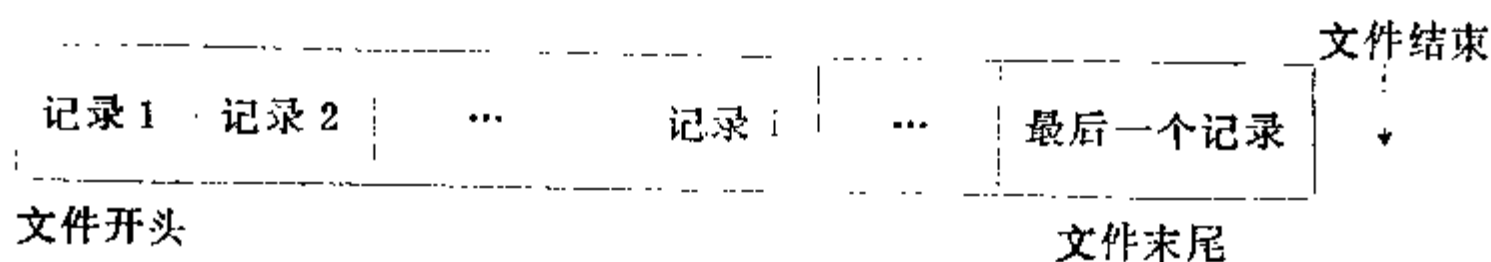
4.4.3 文件的操作使用

要操作与使用文件, 只能通过 PASCAL 语言给出的有关的几个标准过程与标准函数实现。

先说明操作与使用文件中必须了解的几个术语与概念。

1. 文件指针

PASCAL 语言支持的是顺序文件。即对文件的读或者写, 都是以记录为单位顺序进行的。



文 件 F

用来指明读写在文件的哪个位置上进行的一个信息, 被称为文件的指针。文件指针是由计算机系统自动管理的。随着读写的进行, 系统自动修改文件指针。例如, 在读文件 F 时, 若文件指针已指向记录 i, 就把这个记录的内容送到文件的变量缓冲区 F↑中, 并且系统自动修改指针内容, 使它指向下一个记录 i+1。

2. 文件结束

文件的第一个记录是文件的开头, 文件的最后一个记录是文件的末尾。当文件指针指向最后一个记录之后时, 我们就说, 文件已经结束了。文件是否结束, 可用标准函数 EOF 测出。

3. 行结束

对正文 (TEXT类型) 文件, 文件中的全部字符不是简单按顺序排列的, 它们还被划分为若干行。行与行之间用一个结束标志标记, 如用回车 (RETURN)、或回车与换行 (LINE FEED) 两个控制字符隔开。当文件指针指向这个控制字符时, 我们就说, 文件处理到了一行结束的位置。当我们写一个字符序列时, 可以在需要换到下一行的位置, 写入行结束标记。在读文件时, 可以用 EOLN 标准函数测试是否到了行结束位置。不少 PASCAL 语言还支持页结束功能。

4. 打开文件

在使用文件之前，不仅要说明文件，还必须打开文件。从用户的角度看，打开文件的作用是：

第一，指明程序中用的内部文件名对应的真正文件，即建立内、外部文件名之间的对应关系（MS PASCAL 是用另外的办法处理内、外部文件的对应关系的）。

第二，指明将在要打开的文件上执行的操作类别，即是读已有文件，还是新建一个文件，准备写入新内容。

第三，把文件指针移到文件开头，并执行一点附加操作。

打开文件，在 PASCAL 语言中是由 RESET 和 REWRITE 两个标准过程完成的。

5. 关闭文件

在读写完一个文件之后，一般应执行一个关闭文件操作。关闭文件的作用是：

第一、解除内、外部文件名之间的对应关系。这之后，就不能再对相应文件进行任何操作，除非再一次打开它。

第二、如果是写文件，关闭文件操作将使盘上的文件变为永久文件，即程序结束后，这个新写的文件就永久地保存在磁盘上了。否则，程序运行一结束，新写的文件就“消失”了。在对已有文件执行读操作时，不执行关闭文件操作，一般也不会带来新的问题，但在读完之后立即执行关闭文件操作会好些。

执行关闭文件由标准过程 CLOSE 完成。

下面分小段介绍怎么通过使用有关的标准函数和标准过程完成文件的操作与使用。在介绍中，假定 F 是我们要操作的文件的内部文件名。

(1) 判断文件结束的操作

判断文件结束，是调用标准函数 EOF (F) 来完成的。

这是用于判断文件指针是否已移过文件末尾的一个布尔型函数。当指针已移过文件末尾时，EOF (F) 的值为真，否则为假。

在对文件进行读写操作时，了解 EOF (F) 的值是十分重要的，EOF (F) 为假，才能继续读操作，EOF (F) 为真，才能执行写操作。详见后面有关部分的叙述。

(2) 判断行结束的操作

判断文件的行结束，是调用标准函数 EOLN (F) 来完成的。

这是用于判断文件指针是否已移到按行组织的文件的一行末尾的一个布尔函数。当指针已移到一行末尾的时候，EOLN (F) 的值为真，否则为假。

在需要按行来读文件时，了解 EOLN (F) 的值是十分重要的。当然，要把一个文件写成分行的结构，也得有办法把这个行结束的标记写进文件中去。

(3) 打开文件的操作

打开文件操作，由两个标准过程 RESET 与 REWRITE 完成。RESET 用来打开一个已有文件，准备完成读操作；REWRITE 用来打开一个新文件，准备完成写操作。这两个过程在大多数 PASCAL 的编译版本中都有标准与扩展两种格式。

RESET 的标准格式，是 RESET (F)。

这是用于把指针移到一个已经建立起来的文件的开头位置的标准过程。如果 F 文件不是一个空文件的话，它同时还把文件的头一个记录的值传送给 F↑。否则 EOF (F) 为真，F↑ 中内容无意义。在开始读操作之前，必须先执行这个 RESET 过程。

这种格式无法解决内、外部文件名间的对应关系，这个问题要到程序本身之外去解决，不太方便。现在大部分 PASCAL 版本都使用它的扩展方式。扩展了，其格式就不那么统一，但功能基本上是一致的。

MS PASCAL 给出了一个 ASSIGN 过程。它有两个参数，第一个是文件的内部文件名，第二个是文件的外部文件名。ASSIGN 过程的功能是确立内部文件名与外部文件名的对应关系。例如：

```
ASSIGN (F1, 'ABCD.DAT')
```

就把程序内说明的一个文件 F1 与磁盘上的一个叫做 ABCD.DAT 的实际文件联系起来了。在程序执行过程中，凡是对文件 F1 的操作都是对 ABCD.DAT 进行的。

这里的外部文件名，也可以用字符串变量形式给出。

MS PASCAL 语言还给出了一个读入外部文件名的扩展的过程 READFN。其使用格式为：

```
READFN (INPUT, F1);
```

它的功能是从终端读入内部文件 F1 所对应的实际文件的文件名，等同于用 ASSIGN 语句把读入的文件名指定给 F1。

对 READFN 来说，它和 READLN 语句是类似的。区别在于：

- READFN 的第一个参数是应该有的，还必须是文件变量。若不属于这种情况，则认为此处省略了 INPUT 变量名，并给出警告信息。

- F1 是文件变量时，系统将把从文件那读出的字符序列形成的有效文件名指定给 F1。若 F1 不是文件变量，则 READFN 和 READLN 的功能是一样的。

READFN 在系统内部被用来读程序参数，用来读文件名是最方便的。

为文件变量指定实际文件名，指定文件的访问方式，指定文件操作出错时的处理办法等，都必须在打开文件之前进行。文件打开之后，就不能再对该文件指定上述有关内容。我们先在这里进行一点简单说明，到程序实例部分再次说明。

说明文件的访问方式，是把 TERMINAL、SEQUENTIAL 和 DIRECT 指定给文件的 MODE 域。对文件变量 F，就可以用如下方法指定对它的访问方式：

```
F. MODE := SEQUENTIAL    或
```

```
F. MODE := TERMINAL      或
```

```
F. MODE := DIRECT
```

指定文件操作出错时的处理办法，用的是指定文件的 TRAP 域为真或为 FALSE，例如：

```
F. TRAP := TRUE    或
```

```
F. TRAP := FALSE
```

当指定了 F.TRAP 为 TRUE 之后，文件操作一旦出错，出错编号就返回到 F.ERRORS 域中（其值在 0 到 15 之间）。用户可以通过检查该值并依情况进行相应处理。换言之，指定了 F.TRAP 为 TRUE 时，文件操作出错后应由用户本身处理。当指定了 F.TRAP 为假（FALSE）时，文件操作一旦出错，用户程序将立即结束，给出出错信息后，系统返回到操作系统。

打开文件的另一个过程是 REWRITE。REWRITE 的标准格式，是 REWRITE (F)。

这是用于把指针移到一个文件的开头位置，准备往里写新的内容的标准过程。如果这个文件已经存在，则在执行了REWRITE(F)之后，文件原来内容全部丢失，使其成为一个空文件，并使EOF(F)的值变为真。在进行写操作之前，必须首先执行这个标准过程。

(4) 读文件的操作

PASCAL语言中，实现读文件操作的标准过程有两个，它们是GET和READ。GET是用于完成读操作的通用过程，READ则是对用GET读TEXT文件时一种扩展形式。

GET的使用格式是GET(F)，它完成的是把文件F的一个确定的记录的二进制形式的值，原样取到文件变量缓冲区F↑中。具体执行过程如下：

它首先把指针移向文件F的下一个元素，然后把这个元素的值读到F↑中去，这个过程能否正确执行，取决于在执行GET(F)之前，文件F中是否还有下一个记录，若EOF(F)的值为假，则可以执行，否则就是一个错误。

GET标准过程可以读PASCAL语言的任何类型的顺序文件，在标准PASCAL中，READ和READLN标准过程只允许读TEXT类型的文件。

(5) 写文件的操作

PASCAL语言中，完成写文件操作的标准过程有两个：PUT和WRITE。PUT是用于完成写操作的通用过程，而WRITE则是对用PUT过程向TEXT类型文件中进行写操作的一种扩展形式。

PUT的使用格式是PUT(F)，它的功能是把文件变量缓冲区F↑中的内容，以它在程序内存区中的二进制表示形式，原样地写到文件F中去，作为文件的一个新的记录。具体的执行过程如下：

它首先把F↑缓冲区中的元素写到文件F的末尾位置，然后把指针后移一个记录位置。在调用它之前，EOF(F)的值应为真，执行这一过程之后，EOF(F)的值仍然为真。

PUT标准过程可以写PASCAL语言的任何类型的顺序文件，在标准PASCAL中，WRITE和WRITELN标准过程只允许写TEXT类型的文件，使用中要受到一些限制。

(6) 关闭文件的操作

在PASCAL语言中，完成关闭文件操作的标准过程为CLOSE。其使用格式是CLOSE(F)。

这是用于把一个新建在磁盘上的文件关闭，使其变为永久文件的标准过程。在对一个文件进行完写操作之后，必须调用这一过程。

从操作系统的角度讲，它还起着关闭相应输入或输出通道的作用，就是使内部文件与实际文件“脱离联系”。

一般来说，可以不必关闭输入文件。但要反复重读一个文件时，每读完一次就关闭它，等开始下一次读之前，先用RESET过程打开它，可以防止占用太多的输入通道，而使程序无法运行。

已经关闭的文件，在重新打开它之前是不能读写的。

MS PASCAL还给出了另一个关闭文件的过程DISCARD，使用格式是DISCARD(F)。它与CLOSE的区别在于执行它关闭之后，自执行一次对这个文件的删除操作。删除了的文件就不再存在了。

(7) 直接文件的随机读写

前面说过,使用PASCAL语言的顺序文件有两点很不方便的地方,即必须按文件中记录的实际排列次序依次读写,又不能同时对一个文件既读又写。这往往无法满足程序中对文件的使用要求。为此,不少PASCAL版本都扩充了使用顺序文件的功能,增加了对顺序文件的随机(直接)读写功能,即允许用户按指定的记录号读出具体的一个记录,还允许对读出的内容进行修改,再把它回写到文件中原来的位置上。这种扩展表现在两个方面。第一,变原来严格地按次序读写为按记录号直接读写;第二,对同一个文件既可以读,也可以写,而且不再是非写到文件末尾不可,而是写到指定的文件位置。为了实现这一要求,必须首先指明对文件是直接读写,还得能指定记录号。

指明对文件直接读写,是用如下一个赋值语句实现的,

`F.MODE:=DIRECT`

指明记录号是用MS PASCAL语言给出的一个标准过程SEEK。其使用格式是,

`SEEK (F, n)`

n是一个字类型的变量或常量值。SEEK过程的功能是使文件指针指到文件的第n号记录。

读写操作仍是分别用GET (F) 与PUT (F) 过程完成的。

对直接文件,指定了文件名和直接读写方式后,就可以用RESET或REWRITE过程打开它。

要按直接读写方式读指定的一个记录,通常用如下两个语句完成,

`SEEK (F,n); GET (F)`

第一个语句,通过记录号n使文件指针指向F文件的n号记录,第二个语句把该记录的值读到文件变量缓冲区F^中。

要按直接读写方式写指定的一个记录,通常用如下几个语句完成,

`SEEK (F,n);` 把要写的内容填入F^中; `PUT (F)`

第一个语句指定要写的记录号,接下来把新的记录内容填到文件变量缓冲区F^中,然后用PUT (F) 语句把F^中的内容写到F文件的第n号记录的位置。

必须说明,直接文件和SEEK过程都是对标准PASCAL语言的扩展功能,不同机器上的、不同版本的PASCAL编译程序对它们的处理不会全完相同。用户在使用这样的内容时,要查阅有关手册。我们上面讲的是MS PASCAL语言所采用的办法。

在某些PASCAL语言中,直接读写方式不能用于TEXT文件。在MS PASCAL语言中,直接读写方式也允许用于TEXT文件,但规定一个直接读写的记录为TEXT文件中的一行。为此,还必须指定直接读写文件的记录长度,即TEXT文件的最大行长。当不明确给出这个行长时,系统将使用默认的行长(例如80)。此时,SEEK对TEXT文件将设置延缓求值状态为“EMPTY”。

对直接文件的处理与操作过程中,不论EOF (F) 的值为真还是为假,都允许执行GET或PUT操作。用RESET或REWRITE打开的直接文件,都能执行读与写两种操作。当用REWRITE打开直接文件时,若文件尚不存在,它将创建新文件。若文件已存在,文件原有内容也不会丢失。这后一种情况与用REWRITE打开非直接文件是不同的。

使用直接文件遇到的一个困难问题,是文件的准确的结束位置难于确定。尽管EOF

(F)函数是可以使用的,但往往它不能给出文件结束的准确位置。特别是一个已有的顺序的 ASCII 结构的文件,一旦用直接方式修改过它的某些值之后,如果再用顺序方式读写该文件,很有可能出现不能读写它的某些数据的情况。因此,我们建议读者在使用直接文件时,自己设法“记忆”文件的实际长度,以代替 EOF 函数的功能。

4.4.4 TEXT 文件读写中的特殊处理

在我们可以使用的各种类型的文件中,有一种特别常用,特别重要的文件,这就是本节要介绍的 TEXT 文件,又称文本文件,或正文文件。它是由字符组成的文件,就是说 TEXT 文件的成分只能是字符。

这种文件之所以特别重要,主要是由于计算机系统中大部分的输入和输出操作都是以字符形式进行的,它是实现用户和计算机交换信息的方便方式。

为了简化在 TEXT 文件上的输入和输出操作,增强读写语句的功能, PASCAL 语言对 TEXT 文件的读写,规定了一些特殊的处理:

1. 标准的输入和输出文件 INPUT 与 OUTPUT

在前面已经多次提到与用到 INPUT 与 OUTPUT 这两个文件。这是人们事先共同约定的,或者说是 PASCAL 语言默认的标准输入文件与输出文件。就是说,在实现 PASCAL 语言的编译程序时,已用了如下的约定规则:

```
TYPE
    TEXT=PACKED FILE OF CHAR;
VAR
    INPUT, OUTPUT: TEXT;
```

因此,在编写 PASCAL 程序时,无须用户再在自己的程序中对这两个文件进行这样的说明,直接使用它们就行了。

在使用 READ 与 WRITE 语句对这两个文件进行读写操作时,又约定可以省略这两个文件名不写,也就是说,在使用 READ 与 WRITE 语句时,如果未直接给出文件名,则它们将分别在 INPUT 文件与 OUTPUT 文件上进行。

INPUT 与 OUTPUT 一般指的是慢速的,系统中常用的输入与输出设备。在不同的计算机系统中,它们所代表的设备可以不同。在我们使用的 IBM-PC, WANG-PC, 长城 0520 等计算机系统中, INPUT 代表终端键盘输入, OUTPUT 代表终端屏幕输出。

最后要说明的一点是,用户不必对这两个文件执行 RESET(INPUT) 与 REWRITE(OUTPUT) 操作,它们是由系统自动完成的。

2. TEXT 文件读写中自动的数据类型变换

READ 与 WRITE 过程的功能

对 TEXT 文件来讲, READ 过程是对 GET 过程的一种扩展。这种扩展表现在两个方面:

第一、当要读文件中一个字符,并把它赋给一个字符变量 CH 时,应该写为:

```
GET (F), CH:=F↑
```

如果用 READ 过程,则可以直接写

```
READ (F, CH)
```

其意义十分明确，无须做进一步的解释。

第二、当要把一个文件的多个字符读来并依次赋给字符变量 CH1, CH2, ... CHn 时，用 GET 形式的过程就很繁琐，而用 READ 形式的过程就可以简单的写为：

READ (F, CH1, CH2, ..., CHn)

WRITE 过程与 PUT 过程也有类似的关系，即我们可以把 F↑ := CH; PUT (F); 改写成

WRITE (F, CH);

当然也可以用 WRITE (F, CH1, CH2, ..., CHn) 的形式代替一系列赋值和 PUT 操作。

如果文件名不出现在 READ 与 WRITE 过程的参数表中，则表明是到默认的 INPUT 文件去读，要往默认的 OUTPUT 文件中写。这是我们在这之前已多次使用过的办法。否则，文件名一定要作为 READ 和 WRITE 过程的头一个参数。

自动的数据类型变换

如果不是到 TEXT 文件中读取字符，而是读一个整型数据，或实型数据，则必须用 READ 过程，而不能简单地用 GET 过程来完成了。看下面一个例子，如：

READ (F, I, R)

假定这里的 F、I、R 分别为 TEXT、INTEGER 与 REAL 类型的变量，则它表明的是从一个 TEXT 文件 F 中先读来一个整数值，然后再读来一个实数值。I 和 R 在程序变量内存区中，是用二进制形式的值表示的，而文件 F 中登录的是与这个整型值和实型值相对应的字符串。在进行读入的过程中，显然我们不能把 F 文件中的几个字符的 ASCII 码简单的拿来赋给 I 和 R，而必须把它们转换为各自相应的二进制数值。这一转换，是由 READ 过程自动地完成的，无须程序员干预，这就是人们常说的 READ 过程执行时伴有的自动的数据类型变换。我们不在这里做更多的说明，而只是简单地提一下这个事实。

同样，WRITE 过程也有类似的反转换过程。从这个角度讲，READ 与 WRITE 又不是简单地对 GET 与 PUT 过程的扩展，而是加进了新的功能。必须在使用它们的过程中，弄清二者之间的同异，不能混淆。

3. 关于 TEXT 文件中的换行与文件结束的处理

TEXT 文件，当然不是一个简单的字符序列。往往还要把它们若干个字符分成为一行，把若干行又分成一页，既便于计算机的入/出处理，又便于人们书写与阅读。

为此，必须在建立 TEXT 文件的过程中，加进某些行结束或页结束的标记，在整个文件的末尾，还要有个文件结束的标记。

在往一个文件里写入某些内容之后，如一个字符串，表示应结束当前一行，再下面的内容应写在下一行上，可以选用如下两种格式中的一种：

WRITE (F, 'THIS IS A STUDENT');

WRITELN (F)

或者简单地写为：

WRITELN (F, 'THIS IS A STUDENT')

这就在写完这一字符串之后，把一个行结束标志加到了 F 文件中。

关于整个文件的结束处理，不要用户自己去解决，计算机系统在执行 CLOSE (F) 语句时会自动完成。

在读一个 TEXT 文件时，只能随着文件指针的移动逐个字符去读。为了判定是否到了

一行的末尾，或整个文件的末尾，必须通过查询布尔函数EOLN (F) 与EOF (F) 的值才能知道。

为了表示在读完完整的一行之后，或只读了一行中的前面若干字符之后，就要跳过剩下的字符直接到下一行去读，可选用如下两种格式中的一种：

```
READ (F, CH1, CH2, ..., CHn) ;
```

```
READLN (F)
```

或简单地写为：

```
READLN (F, CH1, CH2, ..., CHn)
```

当READ语句中的变量还包括有其它类型数据时，情况也完全一样。

这里讲到的READ和READLN，WRITE和WRITELN的关系，与本书在2.5节的二(2)中提到的是完全一致的，只是这里把它们的使用方式进一步扩展到读写外存上的文件。

下面给出几个使用文件的程序例子。

第一个程序，把 $0^{\circ} \sim 89^{\circ}$ 的每一度的正弦值建成一个文件。

```
PROGRAM FIL01 (INPUT, OUTPUT) ;
VAR
  X, Y: REAL;
  I, LNE: INTEGER;
  SNX: FILE OF REAL;
BEGIN
  ASSIGN (SNX, 'SINX.DAT') ;          REWRITE (SNX) ;
  Y := 3.1415926/180;
  FOR I := 0 TO 89 DO
    BEGIN
      X := I * Y;
      SNX^ := SIN (X) ;
      PUT (SNX)
    END;
  CLOSE (SNX)
END.
```

这个程序中用到的是一个记录为实数类型的文件，用 FILE OF REAL 说明 SNX。

该程序要完成新建文件的操作，所以要用REWRITE⁴过程打开文件。文件的实际文件名作为SINX.DAT。

向实型数据文件写入记录内容，只能用PUT过程实现。

文件建好后，要用CLOSE过程关闭文件，否则程序执行后，建好的文件自动删除。许多初学者常常忘记关闭文件的操作。

第二个程序与第一个类似，但多了个把求得的正弦值送到终端显示和送到打印机打印的操作。终端与打印机都是字符型设备，因此不能用PUT过程向它们传送数据，而只能用WRITE和WRITELN过程。

```

PROGRAM FIL02 (OUTPUT),
VAR
  I:INTEGER;
  X,Y:REAL;
  F1 :TEXT;
  F2 :FILE OF REAL;
BEGIN
  ASSIGN (F1, 'LPT1:');          REWRITE (F1);
  ASSIGN (F2, 'SINX.DAT');        REWRITE (F2);

  X:=3.1415926/180;
  FOR I:=1 TO 99 DO
  BEGIN
    Y:=SIN ( (I-1) * X);
    F2^:=Y;
    WRITE (Y:12:8);
    WRITE (F1, Y:12:8);
    PUT (F2);
    IF (I MOD 5=0) OR (I>89)
      THEN
        BEGIN
          WRITELN;
          WRITELN (F1)
        END
      END;
  CLOSE (F1);
  CLOSE (F2)
END.

```

F1为TEXT文件，对应行式打印机。请注意 REWRITE(F1, 'LPT1:') 的用法。

OUTPUT代表终端输出，不必在变量说明部分再对它进行任何说明。

F2与第一个程序中的 SNX 相同。

请注意，对非TEXT文件，如F2，它的全部记录组成一个简单的数据序列，是没有另外的格式信息的。对这种文件，不能直接送到终端或打印机输出，而只能先用 GET 过程读来每一个记录，再用 WRITE 过程输出。而TEXT文件，除了字符之外，还有一些分行、分页等格式控制信息，这就是这个程序中用 WRITELN 和 WRITELN (F1) 两个语句的原因。

第三个程序，用A·PAS文件复制出一个新文件B·PAS，并在终端上显示A·PAS的内容。给出这个例子的主要目的是想说明使用TEXT文件的常用办法。

```

PROGRAM FIL03 (OUTPUT),
VAR

```

```

    FIN,FOUT:TEXT;
BEGIN
    ASSIGN (FIN, 'A.PAS');          RESET (FIN)
    ASSIGN (FOUT, 'B.PAS');         REWRITE (FOUT);
    WHILE NOT EOF (FIN) DO
    BEGIN
        WHILE NOT EOLN (FIN) DO
        BEGIN
            FOUT := FOUT +
            PUT (FOUT,
            WRITE (FIN);
            GET (FIN)
        END;
        READLN (FIN);
        WRITELN (FOUT);
        WRITELN
    END;
    CLOSE (FOUT)
END.

```

在读文件时，一般用WHILE NOT EOF (FIN) 开始，保证不出现到了文件末尾还给出读操作的错误。

在处理一行输入时，一般用WHILE NOT EOLN (FIN) 判行结束，未遇到行结束，读下一个字符，用 GET 和 READ 都可以，遇到行结束，要执行一个 READLN 操作。

在写TEXT文件时，在需要换行的位置，要用WRITELN写一个换行控制信息到文件中，在向终端输出时也是如此。

第四个程序，完成在132列的打印机上用左边的65列和右边的65列同时打印两个文件，两个文件之间留两个空格。这个程序有实用意义。许多PASCAL语言的初学者不习惯在一行上写几个语句，写出的程序每行都很短，打印时只用132列纸的很窄一条。这时完全可以同时一左一右打两个文件，提高纸张利用率。

这个程序的第一段，要想表明避免用GOTO语句并把程序设计得短一点的想。第二段表明读写TEXT文件过程中的格式控制。在这个程序中，两个文件的行数可以不一样多，第一个文件打印完后继续打印第二个文件时，在第一个文件位置用65个空格代替。第二个文件先结束或一行长不足65时，可以不用打空格而直接回车，这就是那里的FOR语句内的BREAK语句的作用，这就是循环语句的提前退出问题。

```

PROGRAM PRINTF (INPUT,OUTPUT);
VAR
    I,J:INTEGER;
    CH:CHAR;
    LS,NAME:STRING (100)

```

```

F1, F2, F3:TEXT,
BEGIN
  LS:='FIRST',    J:=0,
  REPEAT
    WRITE (' ENTER', LS, ' FILE  NAME: ' );
    READLN (NAME) ;
    IF J= 0
      THEN
        [ ASSIGN (F1, NAME) ;    F1.TRAP:=TRUE,
          RESET (F1) ,
          IF F1.ERRORS<> 0
            THEN
              [ WRITELN ( ' FILE  NOT FOUND, ENTER
                AGAIN! ' ) ;    F1.ERRORS:= 0 ]
            ELSE
              [ J:=J+ 1, LS:='SECOND' ]
          ]
      ELSE
        [ ASSIGN (F2,NAME) ; F2.TRAP:=TRUE,
          RESET (F2) ,
          IF F2.ERRORS<> 0
            THEN
              [ WRITELN ( ' FILE  NOT FOUND, ENTER
                AGAIN! ' ) ;
                F2.ERRORS:= 0 ]
            ELSE J:=J+ 1
          ]
      ]
  UNTIL J= 2 ;
  ASSIGN (F3,'LPT1:') ;      REWRITE (F3) ;
  REPEAT
    FOR I:=1 TO 65 DO
      IF NOT EOF (F1)
        THEN IF NOT EOLN (F1)
          THEN
            [ READ (F1, CH) , WRITE (F3,CH) ]
            ELSE WRITE (F3,' ')
          ELSE WRITE (F3,' ') ;
      IF NOT EOF (F1)
        THEN READLN (F1) ;
      WRITE (F3, ' ') ;

```

```

FOR I:= 1 TO 65 DO
    [ IF NOT EOF (F2)
      THEN IF NOT EOLN (F2)
            THEN [ READ (F2,CH); WRITE (F3,CH) ]
            ELSE BREAK
      ELSE BREAK
    ],
IF NOT EOF (F2)
    THEN READLN (F2) ;
    WRITELN (F3)
UNTIL EOF (F1) AND EOF (F2) ,
CLOSE (F1) ; CLOSE (F2) ;   CLOSE (F3)
END.

```

在打印过程中，两个文件内若有多于65个字符的行，该行第65个字符之后的全部内容不被打印，对其余各行无影响。

思考题：

这个程序的第二段用一个REPEAT语句控制，每重复一次，完成的是什么操作？

用FOR语句控制每个文件的打印宽度方便吗？

在读两个文件的每一个字符时，都用判文件结束和行结束两个条件有道理吗？为什么？

结束打印的条件是怎么给出的？

对两个文件的换行读和对输出文件的换行写是怎么控制的？在开始读第二个文件的一行之前，为什么用了一个WRITE (F3, ' ') 语句？

第五个程序，是对顺序文件进行随机（直接）读写的例子。假定有一个由10名职工的工资等情况建成的文件SALARY.DAT。现在要修改几个人的工资额，可用如下一个程序实现。

```

PROGRAM SALARY (INPUT, OUTPUT) ;
LABEL
    1, 2,
TYPE
    INF=PECORD
        NAME:LSTRING (16) ;
        AGE:15..60,
        SEX: (MALE, FEMALE) ,
        DEPTNO: 1 ..20,
        SALARY:REAL
    END,
VAR
    F:FILE OF INF,   F1:TEXT,
    I:INTEGER,        CH:CHAR,

```



```

    SEXE:STRING (6) ;    REC:INF;
BEGIN
    ASSIGN (F, 'SALARY.DAT') ;          F.TRAP:=TRUE;
    RESET (F) ;
    WRITE ('FILE SALARY.DAT EXIST? (Y/N) ');
    READLN (CH) ;
    IF (CH='N') AND (F.ERRS=0)
    THEN
        BEGIN
            CLOSE (F) ;
            WRITE ('FILE SALARY EXIST, CREATE A NEW
                FILE AFTER') ;
            WRITELN ('DELETING OLD FILE? (Y/N) ');
            READLN (CH) ;
            IF CH='N' THEN GOTO 1
        END
    ELSE
        IF (CH='Y')
        THEN
            IF F.ERRS=0
            THEN BEGIN CLOSE (F) ; GOTO 1 END
            ELSE
                BEGIN
                    WRITELN ('This file not found! ') ;
                    F.ERRS:=0
                END;
            WRITELN ('CREATE SALARY.DAT FILE AND WRITE THE
                CONTENTS. ') ;
            REWRITE (F) ;
            WRITELN ('NAME AGE SEX DEPTNO SALARY') ;
            WRITELN (' ..... ') ;
            FOR I:=1 TO 10 DO WITH REC DO
                BEGIN
                    READLN (NAME, AGE, SEXE, DEPTNO, SALARY) ;
                    IF SEXE='MALE' THEN REC.SEX:=MALE;
                    IF SEXE='FEMALE' THEN REC.SEX:=FEMALE;
                    F^:=REC; PUT (F) ;
                END;
            CLOSE (F) ;

```

```

F.MODE:=DIRECT;  RESET (F) ;
REPEAT
WRITE ('GIVING RECORD NUMBER:');  READLN (I) ;
IF (I>=1) AND (I<=10) THEN
BEGIN
SEEK (F, I) ;  GET (F) ;
WRITELN (F^.NAME:18, F^.SALARY:8:2) ;
WRITE ('NEW SALARY:');
READLN (REC.SALARY) ;
SEEK (F, I) ; F^.SALARY:=REC.SALARY;
PUT (F)
END ELSE
IF I< > 0
THEN WRITELN ('RECORD NUMBER ERROR! ')
UNTIL I=0;
READLN;  WRITELN;
WRITE ('PRINT THE CONTENTS OF SALARY.DAT?(Y/N)');
READLN (CH) ;
IF (CH='Y') OR (CH='y')
THEN
BEGIN
ASSIGN (F1, 'LPT1:');  REWRITE (F1) ;
FOR I:=1 TO 10 DO
BEGIN
SEEK (F,I) ;
SEXE:='MALE';
IF F^.SEX=FEMALE THEN SEXE:='FEMALE';
WRITELN (F1, F^.NAME, F^.AGE:4,SEXE,
F^.DEPTNO:3, F^.SALARY:8:2)
END
END;
WRITELN (F1) ;
CLOSE (F) ;  CLOSE (F1) ;

```

2:

END.

这个程序运行时，首先提示用户给出要修改的记录的记录号，再检查给出的记录号是否在指定的范围（1..10）内。在这个范围内，按记录号读出相应记录内容到F↑中，并显示这个人的姓名和工资数，接着读入新的工资数，并写回文件中。请注意，当发现不需要修改已读出的内容时，把原工资数填在新工资数处就行了，而不能直接按回车键，然后请求输入另一个记录号。当发现记录号不在指定范围内时，检查记录号是否为零。为零，

表示修改完毕，程序执行关闭文件后结束。可以看到，我们是用送入记录号为零表示修改结束意图的。当送入的记录号既不在指定的范围内又不为零时，是操作错误，指明错误后请求继续输入一个新的记录号。

给出这个程序例子的目的：第一个目的是说明顺序文件的随机读写，其方法从程序清单可以看清：① 打开文件前，必须指定文件为 DIRECT 读写方式，② 要用 SEEK 指定记录号并用 GET 过程读出记录，③ 把要修改的内容写入 F↑中，这里是用 READLN 语句实现的，当然也可以用赋值语句实现。④ 用 PUT 语句把修改后的内容写回文件刚用 SEEK 指定的那个记录的所在位置。最后要关闭文件。使用这种办法处理文件的好处是明显的，此时，只要把文件中需要修改的记录读出来，修改后再写回就可以了，涉及不到其余的记录。不用这种办法，就只能复制整个文件，在复制过程中，把要修改的有关记录修改好，其余记录原样复制。当文件中记录个数比较多时，复制文件是很浪费机器时间和外存空间的一件事，还容易出现读写错误。当然，这还没解决在顺序文件当中删除或插入一个记录的问题。在实际应用中，可以采用在文件记录中增加一个删除标记的办法。在删除时，把要删除的记录加上这个标记，而不是真正删除它。在以后的读文件过程中，对读出内容要先检查这个删除标记，以判别读出数据的有效性。当到了一定时间，文件中这种被删除的记录数目较多时，再用复制文件的办法对整个文件处理一次。在顺序文件当中插入一个新记录是一件困难的事。当对文件中的记录次序没有特殊要求时，可以把新记录一律写在文件末尾，这很简便。当记录的次序必须满足一定条件时，就会遇到把新记录插入到文件当中某个位置的问题。当要插入的新记录个数不多时，也可以先把它们放在文件末尾，以后用到时，要记住这件事，对它们特殊处理，等有了合适的机会（要复制文件时）再把它们插入合适位置。也可以把这些新记录建成一个新文件，以后与原有文件同时使用。当要插入的新记录个数很多时，最好把新的记录先建成一个新的顺序文件，然后对新旧两个文件的全部记录进行排序，以得到一个最终文件。

给出这个程序例子的第二个目的，是说明文件记录为记录型数据时的用法。这也正是 PASCAL 语言相对于别的语言，如 FORTRAN, BASIC 等很有特色的用法，也是用 PASCAL 语言编写事务管理等软件必然用到的方案。在这里，一是要在说明文件时给出文件的记录类型的完整说明。例如把一名职工的情况说明为姓名、年令、工作部门编号、工资等。在以后读写一个文件记录时，每次就能处理一个人的完整信息，此时，往往要用到 WITH 语句处理这个记录类型的数据，从程序清单上可以看到这种用法。必须说明，这个程序执行的是建立与修改已有文件的内容。建立这样的文件与建立实型文件、整型文件等是类似的，主要差别在于文件记录的类型不同。

在实际应用中，必须注意以下三个问题。

首先是文件在磁盘上存放的数据格式。一般应以数据在计算机内的表示形式（二进制形式）存放为好，既省外存，输入输出不再需要进行数据类型变换，速度快，而不宜直接以字符形式存放。因为难于对以字符形式存放的文件进行随机读写，使修改文件的记录内容变得困难起来（实际应用中却常常有这种要求）。从另一个方面看，当文件记录为记录类型，而记录类型中有的域又属于枚举类型（如 SEX: (MALE, FEMALE);) 时，对这种数据是不能用 READ 和 WRITE 语句处理的（前面多次提到）。所以，应该用 PUT (F) 语句把文件的记录写入文件，此语句能把各种合法记录写入文件，这样，外存上保存的数据就是二进制格式的。

第二，对用PUT (F) 写入非TEXT文件中的记录，只能用GET语句读出。SEEK语句只能用于为用随机读写方式打开的直接文件指定记录号。

第三，要简单方便地修改文件中部分记录的内容，应首先指定文件为直接读写方式，用SEEK指定记录号，用GET过程读指定的记录，修改相应内容后再用PUT过程写回到文件中原来位置，并不在SEEK与GET语句之间、SEEK与PUT语句之间进行对该文件的读写操作。

4.5 地址类型数据及其引用

地址类型是MS PASCAL语言对标准PASCAL语言的扩充功能。引入地址类型的目的，是希望能直接通过数据在内存中的地址来引用数据。前边已经看到，标准PASCAL语言总是通过变量名来引用静态变量，这些变量和它们在内存中的地址分配，是由编译程序在编译期间解决的，用户没有办法在自己的程序中加以改变。这对于简化程序设计，提高程序的设计质量是很有利的。但同时也应该看到问题的另一个方面，就是上述办法，对用户设计程序和使用内存所加的限制是十分严格的，有时反而使用户感到不方便。设立地址类型的目的，就是要“突破”这种限制，使用户能“随意”地分配与安排一片内存中的内容，并通过地址变量，来引用该内存区中的有关内容，这很类似于汇编语言程序设计中的常用的手段。这在实现某些系统程序的设计时是很必要的，也会给某些应用程序的设计带来很大的灵活性。这样做带来的一个问题是，用这种办法引用数据是不够“安全”的。原因很简单，冲破PASCAL语言的某些限制得到的灵活性，是以避开PASCAL编译程序的某些语法检查和语义检查而获得的，这些检查此时就落到了用户自己的头上，用户在自己的程序中很有可能用错某些内容。一些刚接触PASCAL语言程序的读者，要特别当心地使用地址类型。

下面详细介绍地址类型的定义和使用问题。在CPU芯片为8086/8088的微机系统中，内存地址被分成两种。一种被称为段际地址 (Segmented Address)，是用连续的两个内存字，即32位二进制表示的。其中一个字 (在高地址) 用于存放段号，即段寄存器值 (Segment Register Value)，另一个字 (在低地址) 用于存放段内的地址偏移值 (Relative Offset Value)。程序在访问非默认的数据段中的数据时，是必须使用这种地址的。另一种被称为段内的地址，是用一个内存字，即16位二进制表示的。这种地址是在程序中使用默认的数据段中的数据时要使用的。

MS PASCAL语言给出了两个关键字ADS和ADR，它们既是定义地址类型的前缀关键字，也是执行语句中的地址操作符。在说明部分，定义地址变量必须使用地址类型的前缀关键字ADS和ADR。例如：

```
VAR
    A:    ADS    OF    类型标识符;
    B:    ADR    OF    类型标识符;
```

冒号左边的标识符A和B，就被定义成引用属于相应类型数据的段际地址和段内地址的变量。用ADS定义的是段际地址类型的变量，用ADR定义的是段内地址类型的变量。

在程序的执行体中，要用地址操作符ADS和ADR取得一个变量或相应常量的地址。例如：

A := ADS 变量, B := ADR 'ABCD';
B := ADR 变量;

这里的标识符A和B必须是前面已说明过的地址变量,而且ADS、ADR后面跟的变量(或常量)必须属于说明地址变量A、B时用到的类型标识符规定的类型,否则编译程序会指出一个赋值语句赋值号左右数据类型不兼容(相容)的错误。

段内地址变量,也可以被看成由唯一的一个域组成的记录类型的变量。如B也可以被写成B·R,此时B·R与字类型变量是兼容的。要表示地址变量B所指向变量的内容,要用B↑(注意,不能用B·R↑)。

类似的,段际地址变量也可以被看成由两个域组成的一个记录类型的变量。如A就可以被写成A·S和A·R两个域变量。这两个域变量都与字类型变量是兼容的。要表示地址变量A所指向的变量的内容,要用A↑(不能用A·S↑或A·R↑,因为A·S和A·R此时是一个段际地址变量的一个域,分别代表段际地址的段寄存器值和段内地址偏移值,不能单独用于引用变量)。

必须说明,任何变量的段内地址变量都属于相同的类型。任何变量的段际地址变量也都属于相同的类型。相同类型的变量就可以彼此赋值。因此把用ADR OF WORD定义的变量C赋给用ADR OF STRING(20)定义的变量D,即D:=C是合法的。用ADS定义地址变量与用ADR定义的变量是不兼容的,二者不能彼此赋值,但二者的·R的域分量仍然是相同的类型。

在说明过程的变量形式参数时,可以用VARS和CONSTS(唯读变量形参)作为变量形式参数的前缀,表示此处的变量地址要用段际地址。调用过程时,调用程序中给出的实际参数就可以不在调用程序的默认数据段中(用VAR与CONST作为前缀的变量参数的实际参数,只能是在调用程序的默认数据段中的变量)。

必须看到,地址类型和指针类型在说明和使用上都是完全不同的。地址类型是面向机器内存地址的,它总是与内存的实际地址相对应。所以使用了地址类型的PASCAL程序移植性就很差。而指针类型,是一种抽象的数据类型,在各种计算机中都是用相同的办法管理的。用指针引用的变量的地址不要求一定对应哪一个实际内存地址,只要满足程序的使用要求即可。

MS PASCAL语言给出了两个预先定义地址类型,它们是:

ADRMEM=ADR OF ARRAY[0..32765] OF BYTE;

ADSMEM=ADS OF ARRAY[0..32765] OF BYTE;

由于它们是字节类型的数组的地址类型,因此可以用使用数组的办法来取得这个数组中的每个字节的值。

再次强调说明,引入地址类型的目的,是通过它引用其它变量。即我们真正要处理的对象是那些有关变量,而不是地址类型的变量本身。

地址类型变量的·R分量和·S分量都与WORD类型兼容,它们可以作为字类型量参加表达式运算,也都可以接收结果为字类型的表达式的计算结果,这在使用地址变量引用一些数据时是必要的。

ADR和ADS在表达式中是一元操作符,即是地址操作符。作为操作符,它后面只能跟变量和常量,而一定不能跟一个表达式。

通过地址类型变量引用数据的格式,是在地址变量名后跟上一个字符'↑'(或'Λ')。

例如A↑、B↑。这里的A和B是地址类型的变量，A↑和B↑则表示由A和B所指向的变量，换句话说，A和B给出的是变量的地址。当A和B指向的数据为记录或数组等结构数据时，要表示该记录的C域、或该数组的第i个元素，要用A↑.C和B↑[i]的格式，即字符'↑'要直接跟在地址变量之后，再接下去给出域名或数组下标。在使用中，不能把三者的次序用错了。

下面我们将用几个程序实例进一步阐明上述基本内容。地址类型，是与具体计算机硬件中的地址管理，以及某些系统软件直接有关的。因此，使用了地址类型的PASCAL程序，是很难在不同系列的计算机系统之间移植的。

第一个程序，用来说明定义和使用地址变量的基本方法，和表明属于ADR类型的地址变量都彼此兼容的概念。这个程序本身无具体的使用价值。

程序清单和这个程序的运行结果如下：

```

program adr (input, output);
var i:integer;                a: adr of integer;
    p1: adr of string (200);  b: adr of real;
    p2: adr of integer4;
    p3: adr of array [1..50] of integer4;
begin
    p2^:=16#01020a0b;        p1:=p2;    a:=b;
    write (p2^:12);
    for i:=1 to 4 do
        write (ord (p1^[i]):6); p3:=p2; write (P2↑:16);
    writeln;
    for i:=1 to 50 do p3^[i]:=12;
    p1:=p3;
    for i:=1 to 16 do write (ord (p1^[i]):4);
    writeln;
    for i:=1 to 4 do write (p3^[i]:8)
end.

```

```

          16910859      11      10          2      1      00000001020A0B
12      0      0      0      12      0      0      0      12      0      0      0      12      0      0      0
          12          12      12          12

```

在这个程序中，说明了5个ADR类型的地址变量。它们被用来引用五种不同类型的数。说明地址变量的格式，都是用在ADR of之后给出一个类型标识符(如INTEGER, INTEGER4或REAL)或一个类型的具体定义(如STRING(200)、ARRAY [1..50] OF INTEGER4)。这是说明ADR类型的地址变量的通用方法，说明ADS类型的地址变量的方法与此类似，只要把这里的ADR换成ADS就可以了。

前面说过，属于ADR类型的地址变量都彼此兼容。在程序中，就有p1:=p2; a:=b; p3:=p2和p1:=p3四个赋值语句，这都是允许的。它们实现的是把赋值号右边的地址变量中的地址值赋给赋值号左边的地址变量。对ADR类型的地址变量来说，p1:=p2语

句和 $p1.r := p2.r$ 语句完成的功能完全相同,但是和 $p1 \uparrow := p2 \uparrow$ 绝对不是一回事。在该程序中,这最后一个赋值语句是错误的,赋值号左右的数据类型不兼容。

在使用地址变量的方法上,本程序中有一个很大的错误,就是地址变量 $p2$ 和 b 本身尚没有赋初值就被拿来使用了,这是没有道理的,这与使用尚无初值的静态变量是相同的错误。这个错误对本程序的运行结果没有产生影响。

这个程序的功能,第一小段完成把一个用 16 进制表示的数值赋给地址变量 $p2$ 引用的 $INTEGER4$ 类型的变量,然后把 $p2$ 的内容赋给 $p1$,也就是把一个 $STRING(200)$ 的变量的起始地址安排在一个 $INTEGER4$ 变量的地址位置。最后输出变量 $STRING(200)$ 的前四个字符位置上的内容,实际上,也就是把 $INTEGER4$ 占用的四个字节的每个字节的内容显示出来。把这一输出结果与 $INTEGER4$ 的按 16 进制输出的结果比较,会发现 $INTEGER4$ 类型变量的内部表示,是按高位字节在内存的低地址的次序安排的,与字符串的安排方式正好相反。

这个程序的最后两小段,是使 $STRING(200)$ 的起始地址与程序中的数组的起始地址为同一个地址,并分别按 $INTEGER4$ 的数组和紧缩字符数组的方式输出同一片内存区中的内容。其结果已在程序清单的后面给出。

从这个程序中可以清楚地看到,有了地址类型,就能把同一片内存区作为存放不同类型数据的存储区域使用。这种用法在后面的程序中会看得更清楚。

第二个程序,是用来表明变量本身和它的地址之间的关系的,顺便看一看程序中的静态变量在内存中是如何安排的。

程序清单和它的运行结果给出如下:

```

program addr [input, output] ;
var  i, j, l:integer;          iadr: adr of integer;
    p, r: real ;              radr: adr of real ;
                                cadr: adr of char ;

begin
    i:=1234; j:=24000;         iadr:=adr i ;
    writeln (' address of I : ', iadr, r:-8) ;
    writeln (i:12, j:12, iadr^:12) ;
    iadr^:=i ;                 writeln i:12, j:12);
    writeln ;
    p:=12.456; r:=456.789;     radr:=adr r ;
    writeln (p:12:4, r:12:4, radr^:12:4) ;
    radr^:=p*r-100;            writeln;
    writeln (p:12:4, r:12:4, radr^:12:4);
    writeln (' address of R : ', radr, r:-8);
    radr.r:=iadr.r ;           writeln;
    writeln (radr^:12:4, ' radr.r=', radr.r:-8) ;
    cadr.r:=radr.r;            writeln;
    for l:=1 to 4 do
        [ write(ord(cadr^):4);   cadr.r:=cadr.r+1 ]

```

ena.

该程序的运行结果:

```
address of I      :    61630
      1234          24000    1234
      24000          24000
      12.4560       456.7890    456.7890
      12.4560       5589.7640   5589.7640
address of R      :    61642
1732681000000000000.0000   radr.r=61630
192    93    192    93
```

在这个程序中,说明了三个整型变量和一个引用整型量的ADR类型的地址变量。说明两个实型变量和一个引用实型量的地址类型的地址变量,此外还说明了一个引用字符变量的ADR类型的变量。

在程序的执行部分,有两个地址赋值语句, $iadr:=adr\ i$ 和 $radr:=adr\ r$, 它们的作用是把变量 i 和 r 的在内存中地址分别赋给地址变量 $iadr$ 和 $radr$ 。这是取得变量地址的通用方式。这之后,我们就可以通过 $iadr$ 和 $radr$ 来引用变量 i 和 r 的值了, $iadr↑$ 和 i , $radr↑$ 和 r 所表示的是同一个内容。这从输出结果的第2行和第4行中可以看到。

我们也可以在赋值语句中,向地址变量所引用的变量进行赋值,例如, $iadr↑:=j$ 和 $radr↑:=p*r-100$ 等语句就是这样用的。赋值的结果可以从运行结果的第三行和第五行中看清。这两个语句的功能分别和 $i:=j$ 与 $r:=p*r-100$ 是一样的。

程序的最后一小段,是让 $cadr$ 取 $radr$ 的值,并把一个实型量占用的四个字节的价值作为整型量输出出来。

下面说明程序中的变量在内存中的安排问题。编译程序将依变量说明的次序,从内存中某一地址开始,按从低向高的顺序逐一分配每个变量的地址值。按此说法,我们可以把程序中的变量的安排关系表示如下:

i	j	l	iadr	p	r	radr	cadr
---	---	---	------	---	---	------	------

每个变量应分得几个字节,是按有关规定执行的。如此处的 p 和 r 各用四个字节,其它变量各用两个字节。

每个变量的地址值是可以输出的。对ADR类型的地址变量,要输出它的 r 分量(与字类型兼容),而不是地址变量本身。这在程序中看得很清楚。这就是程序运行结果中第一行和第六行给出的地址值。需说明的是,第一行中给出的地址值可能会变的。程序中 $iadr:=adr\ i$ 的作用就是把 i 的地址写进 $iadr$ 单元,使 $iadr$ 指向变量 i 。对 $radr$ 和 r 也有同样的作用。我们可以把程序开始后的某一时刻这片内存区中的值表示如下:

↓			↓		↓		↓
24000	24000	未定	61630	12.456	456.789	61642	未定
i	j	l	iadr	p	r	radr	cadr

我们检查一下各变量的地址值,它们是:


```

i      61630                (iadr.r)
j      61630 + 2 = 61632
l      61632 + 2 = 61634
p      61634 + 2 = 61636
p      61636 + 2 = 61638
r      61638 + 4 = 61642 (radr.r)
radr   61642 + 4 = 61646
cadr   61646 + 2 = 61648

```

这与我们程序的输出结果是吻合的。

在使用地址类型时，了解变量在内存中的安排次序，和它们占用的存储单元数目是很必要的。

第三个程序，是把512个字符组成的紧缩数组的前边若干个单元，也当作其它几种类型的数据使用的例子。这个程序中，还给出了对地址进行计算的使用方法。

下面是这个程序的清单。

```

program addr (output);
type str=string (512);
var
  i : integer;      r:real;  c:char;
  buf: str;
  bufadr:adr of str;

  iadr : adr of integer;
  i4adr: adr of integer4;
  radr : adr of real;
  r8adr: adr of real8;
  cadr : adr of char;

begin
  bufadr:=adr buf;  cadr.r:=bufadr.r;
  iadr.r:=bufadr.r;  i4adr.r:=iadr.r;
  radr.r:=bufadr.r;  r8adr.r:=radr.r;
  writeln (bufadr.r,cadr.r,iadr.r,r8adr.r);

  fille (adr buf, 512, 'A');
  for i:=1 to 40 do [ write (cadr^);  cadr.r:=cadr.r+1];writeln;
  for i:=1 to 10 do [ write (iadr^:6); iadr.r:=iadr.r+2];writeln;
  for i:=1 to 5 do [ write (radr^);  radr.r:=radr.r+4];writeln;
  writeln (i4adr^:14, r8adr^:20:10);  cadr.r:=bufadr.r;
  i4adr^:=16#7 BCD 1234;  writeln (i4adr^:16);
  writeln (i4adr^:14, r8adr^:20:10, iadr^);

  for i:=1 to 10 do [ write (cadr^);  cadr.r:=cadr.r+1];
  writeln;  cadr.r:=bufadr.r;

```

```

        for i:=1 to 10 do [ write (ord (cadr↑):4); cadr.r:=cadr.r+1; ]
end.

```

程序的执行结果如下:

```

        61112          61112          61112          61112
AAAAAAAAAAAAAAAA\AAAAA\AAAA\AAAA\AAAA\AAAA\AAAA\AAAA\AAAA\AAAA\
16705 1670 16705 16705 16705 16705 16705 16705 16705 16705
1.2078430E+01 1.2078430E+01 1.2078430E+01 1.2078430E+01
1.2078430E+01
1091795585 2261634 5098639000
0000007BCD1234
2077037108 2261634.9671958000 16705
4(AAAAAA
52 18 205 123 65 65 65 65 65 65

```

在程序执行体开始位置给出的几个地址赋值语句,用于把变量BUF的起始地址,赋给六个ADR类型的地址变量。这里只有第一个语句是必须以bufadr:=adr buf的方式给出。其余五个赋值语句中的.r都可以取消,例如把iadr.r:=bufadr.r写成iadr:=bufadr也是对的。

fillc (adr buf, 512, 'A') 完成把buf的512个单元全填充为字符A。

下面的三个输出语句,执行的是把 buf 的前若干个单元的内容当作字符、整型量和实型量进行输出。输出过程中,按变量占用的存储单元数修改地址变量的值。从这里看到用同一个地址变量引用一片内存区中多个数据的方法。

下一小段是把buf的前四个单元的内容,作为INTEGER4类型的数据进行输出,把前8个单元的内容作为REAL 8 类型数据进行输出的例子。然后用向 i4adr↑赋值的办法修改 buf前四个单元的内容,再重新输出。从程序结果上看到,此时 i4adr↑和r8adr↑的值已经变化了,读者可思考一下为什么iadr↑的值没有改变;在向 i4adr↑赋值之前的 cadr.r:=bufadr.r语句,对最后一小段程序的执行有什么作用。

第四个程序,是把512字符组成的紧缩数组,也当成四种不同记录类型构成的数组使用的例子。这个程序所用的办法有一定的代表性,稍加改进,就可以成为有实用价值的程序。这个程序清楚地表明,说明数据变量和说明地址类型变量的关系。说明了一个数据变量,如程序中的 i、r、c、w 和 buf 等,就要在内存中为它们分配相应的存储单元。而说明一个 ADR 类型的地址变量,只要在内存为它们分配两个字节的存储字就够了,而不是它所引用的数据的一片存储区。只要我们使地址取得相应的地址值,在程序中就可以从这个地址开始的一片内存区当作该地址变量所引用的数据区使用。

下面是这个程序的程序清单。

```

program addr (input, output);
type str=string (512);
var
    i: integer; r: real; c: char; w: word;
    buf: str;
    bufadr: adr of str;

```


12593 12593

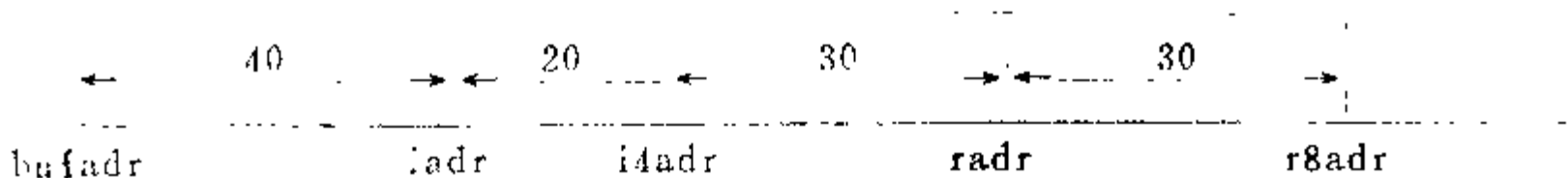
输出长整型数数组的两个域的值

```
12593      825307111
12593      825307111
12593      825307111
12593      825307111
12593      825307111
```

输出实数数组的两个域的值

```
12593      2.5784850E+09
12593      2.5784850E+09
12593      2.5784850E+09
12593      2.5784850E+09
12593      2.5784850E+09
```

第五个程序与第四个程序有类似之处。区别在于，在第五个程序中，把buf的前40个字符作为紧缩字符数组使用。把接续下来的20个、30个和再30个字符分别当作三种数组的前五个元素的存储单元使用。内存分配如下：



程序开始时，先确定了每个地址变量的初值。然后使buf的每个字符被赋值成字母M。接下来，是对buf的不同区域执行赋初值和输出操作。

程序的最后，是对赋过初值buf区域每个字符位置的内容执行输出操作。此时输出的是每个字符位置的整型值，否则，某些位置上的内容不是可打印字符的编码，无法用输出字符的方式得到易懂的正确结果。这里的77是字母M的编码、97是字母a的编码，其它码值，是不同类型的数值的各个字节位置上的数值。对整型数，这些数值的意义是容易理解的，而对实型量和长实型量的每个字节上的值，就不大容易与实型量、长实型量的取值相对应了。从结果中，很容易发现buf的变化情况。请思考，为什么在显示的buf的200个元素中，有两部分保持77编码值。

从程序中还可以看到，使用地址变量时，对引用的数据的地址是不进行“边界”检查的。通过不同地址引用的数据在内存中是可以相互“复盖”的。其实，在前边的几个程序中我们已这样用过了，在这里，我们不过是把这个问题再明确地重申一次而已。

下面给出了这个程序的程序清单和程序的运行结果。

```
program addr (input, output);
type str = string (512);
var
  i      : integer;      r      : real;      c      : char;      w: word;
  buf:str;
  bufaor : adr of str;
  iadr   : adr of array [1..120] of
    record prt: integer; key: integer end;
  record prt: integer; key: integer end;
```

```

i4adr : adr of array[1..80] of
    record prt: integer ; key: integer4 end;
radr : adr of array[1..80] of
    record prt: integer; key: real end;
r8adr : adr of array[1..48] of
    record prt: integer; key: real8 end;
begin
    bufadr:=adr buf ; w:=bufadr.r;
    iadr.r:=w+40 ; i4adr.r:=iadr.r+20 ;
    radr.r:=i4adr.r+30; r8adr.r:=rad.r+30 ;
    writeln ('buf 的起始地址 : ',bufadr.r, w) ;
    fille (adr buf, 512,'M') ;
    for i:=1 to 40 do buf[i]:='a';
    writeln (' 输出紧缩字符数组每个元素的值') ;
    for i:=1 to 40 do write (buf[i]); writeln ;
    writeln;
    for i:=1 to 5 do with iadr^[i] do
        [ prt:=i ; key:=i*10 ];
    writeln (' 输出整型数数组的两个域的值') ;
    for i:=1 to 5 do writeln (iadr^[i].prt:8, iadr^[i].key:8) ;
    writeln;
    for i:=1 to 5 do with i4adr^[i] do
        [ prt:=i ; key:=i+30 ];
    writeln (' 输出长整型数数组的两个域的值') ;
    for i:=1 to 5 do writeln (i4adr^[i].prt, i4adr^[i].key) ;
    writeln;
    for i:=1 to 5 do
        [ r8adr^[i].prt:=i; r8adr^[i].key:=400.2*i ];
    writeln(' 输出长实型数数组的两个域的值') ;
    for i:=1 to 5 do writeln (r8adr^[i].prt, r8adr^[i].key) ;
    writeln;
    writeln (' 再次输出紧缩字符数组每个元素的值') ;
    for i:=1 to 200 do
        write (ord (buf [i]) :4) ;
    end.

```

buf 的起始地址: 00968 00968

输出紧缩字符数组每个元素的值

aa

输出整型数数组的两个域的值

1 10

2	20
3	30
4	40
5	50

输出长整型数数组的两个域的值

1	31
2	32
3	33
4	34
5	35

输出长实型数数组的两个域的值

1	4.002000E+002
2	8.004000E+002
3	1.200600E+003
4	1.600800E+003
5	2.001000E+003

再此输出紧缩字符数组的两个域的值

97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97
97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97	97
1	0	10	0	2	0	20	0	3	0	30	0	4	0	40	0	5	0	50	0
1	0	31	0	0	0	2	0	32	0	0	0	3	0	33	0	0	0	4	0
34	0	0	0	5	0	35	0	0	0	77	77	77	77	77	77	77	77	77	77
77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77
1	0	51	51	51	51	51	3	121	64	2	0	51	51	51	51	51	3	137	64
3	0	102	102	102	102	102	194	146	64	4	0	51	51	51	51	51	3	153	64
5	0	0	0	0	0	0	68	150	64	77	77	77	77	77	77	77	77	77	77
77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77	77

4.6 指针类型

在PASCAL语言中，指针类型与动态数据是密切联系在一起的。

所谓动态数据，是指在程序运行的过程中，能按需要随时建立起来，并可以随时撤消的数据。这与我们前面已经讲过的各种类型的数据有极大差别。那些数据，允许有各自的变量名，并只能通过变量名引用它们。它们的名字、类型、数据个数（文件除外），在程序投入运行之前就已确定，在程序运行过程中也不变化，唯一能改变的是它们的取值，因此，我们称它们为PASCAL语言的静态数据。

动态数据则不同。在程序运行之前，只定义了它的类型和用于指向它的指针变量。动态数据没有自己的变量名，因此不能经过名字访问它们，一类动态数据的数量多少，也不是在用户程序中直接给出的，要由程序运行情况确定。

为什么要使用动态数据呢？原因有二。首先，有些特定结构的数据，主要是用递归方

式定义的数据，如树形结构数据，各种链表结构数据，用动态数据表示它们有很大的方便性，访问数据或修改数据之间的联系十分简单与高效。其次，如果要在一个程序里使用数目不详，多少相差又十分悬殊的数据时，使用动态数据能大大改善程序的质量。所以，能正确的使用动态数据，是提高PASCAL语言程序设计水平的重要一环。

使用动态数据，要解决以下两个问题：

怎样建立与撤消一个动态数据；

怎样实现已经建立起来的一批同类动态数据之间的联系。

本节将围绕以上两个问题进行讲解。

4.6.1 指针的概念和指针说明

指针 (POINTER) 是一种简单数据类型。顾名思义，指针是指明一个数据用的有关信息。联想一下，找一个人，可以告诉他的名字，通过名字去找到这个人；也可以告诉他家的地址，通过地址去找到这个人。这个地址就是找人用的一个指针。同理，访问一个数据，也有类似的两种办法。简单地说，指针是记录一个数据的地址用的有关信息。

在PASCAL语言中，定义指针类型的格式如下图所示：

——标识符—— = ———— —↑—— 类型标识符——□

其中：

标识符是被定义的指针类型的标识符，由用户确定，必须符合标识符的构成规则。

类型标识符，可以是已经被定义过的，但实际使用中，一般是尚待定义的一种数据类型，这是PASCAL语法中，允许先使用而后说明的一个符号的特例之一。它指出的是通过指针变量引用的数据的类型。

↑是定义指针用的特定字符，正是通过这个字符'↑'来表明这里定义的是一个指针类型。不少计算机系统中用字符'∧'代替字符'↑'。

例如：

TYPE

LINK = ↑ DATA;

我们可以把这一说明读作为：定义LINK是指向DATA类型数据的指针类型。这里的DATA是一个类型标识符。上面已经提到，它往往是一个尚待定义、多为记录的数据类型。因此，接下来可以定义DATA本身，例如：

DATA = RECORD

NEXT: LINK;

INT: INTEGER

END;

可以看到，定义DATA时用到了类型标识符LINK（这是最普遍、最重要的用法），而前面定义LINK时，又要用DATA。这是按对任何一个符号都要先说明而后使用的一般规则无法解决的矛盾。为此，PASCAL语法才规定，在这种场合，要先定义指针类型，再定义有关的类型标识符。程序设计人员不能倒换这两者的说明次序。

有了上述的类型定义，就可以用它们来说明指针变量了。例如：

VAR

P, Q, R: LINK;

有几个概念和符号必须弄清楚。

(2) 指针本身的内容是确定类型的一个数据在内存堆区 (HEAP) 中的地址, 一般情况下, 这个内容可以在用 NEW 过程建立动态数据的过程中, 由计算机系统产生, 也可以用赋值语句, 在相同类型的指针之间通过赋值得到, 如 $P := Q$ 。为了表示一个指针不指向任何一个数据, 应该向它赋一个空值, 如 $P := \text{NIL}$ 。这里的 NIL 是指针变量的一个常量值, 是 PASCAL 语言的保留关键字, 可以把它赋给任何一个指针。不同类型的指针之间不能彼此赋值。指针本身的内容是用它的变量名表示的。

请注意↑DATA, P 和 P↑二者间的关系和各自代表的含义, 不能混淆。

从图上可以看到, 指针本身之间的赋值, 如 $P := Q$ 是把一个指针中保存的地址赋给另一个指针。而 $P \uparrow := Q \uparrow$ 实现的是把一个指针所引用的变量的内容传送到另一个指针所引用的变量中去, 指针本身的内容并不变化。但请注意, 这两种赋值操作都只能在相同类型的指针之间进行, 而指针常量 NIL 可以被赋给任何一个指针变量。

(i) 开始时的情形:

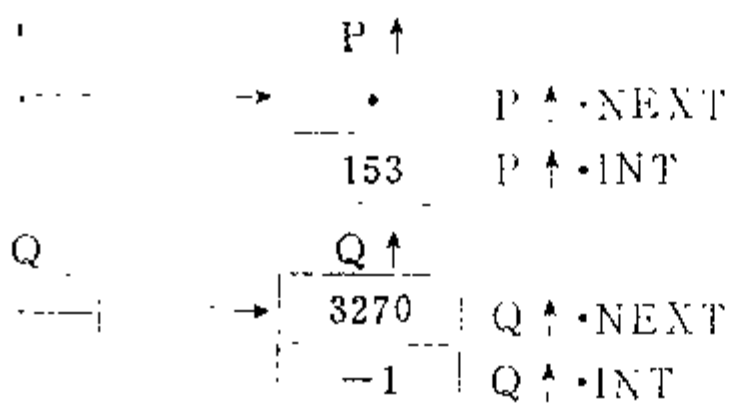


Diagram illustrating pointer manipulation:

- Node 153: P → 153 (已无法访问)
- Node 3270: Q → 3270
- Node 3270 points to -1.
- Node 153 points to Node 3270.

P	$\xrightarrow{\quad}$	$\frac{P \uparrow}{3270}$	$P \uparrow \cdot \text{NEXT}$
		-1	$P \uparrow \cdot \text{INT}$
Q	$\xrightarrow{\quad}$	$\frac{Q \uparrow}{3270}$	$Q \uparrow \cdot \text{NEXT}$
		-1	$Q \uparrow \cdot \text{INT}$

4.6.2 动态变量的使用方法

使用动态变量的两个问题是，如何建立与撤消一个动态数据，又如何建立一批动态数据之间的联系。

在PASCAL语言中，建立一个动态数据，是通过调用标准过程NEW实现的；撤消一个动态数据，是用标准过程DISPOSE完成的。建立一批动态数据之间的联系，是借助数据内的指针来解决的。过程NEW和DISPOSE的参量(ARGUMENT)只能是指针。在MS PASCAL语言中，若参量指针为动态数组类型时，NEW和DISPOSE的参数中还可以包括另外一点附加信息。

一、建立一个新的动态数据

建立一个动态数据，是用NEW过程完成的，例如NEW(P)，它的作用，是要建立一个用指针P引用的动态变量。说得更准确些，NEW(P)执行过之后，一是从内存堆区分配出一块正好放下相应数据的存储区，二是把该存储区的首地址写入P中，这就使指针P指向了这个存储区。但必须强调指出，此时的P↑，即这个新建立动态数据的内容尚未确定，还必须用另外一些语句向它的各个域进行赋值，例如：

P↑.NEXT:=NIL;

P↑.INT :=2000;

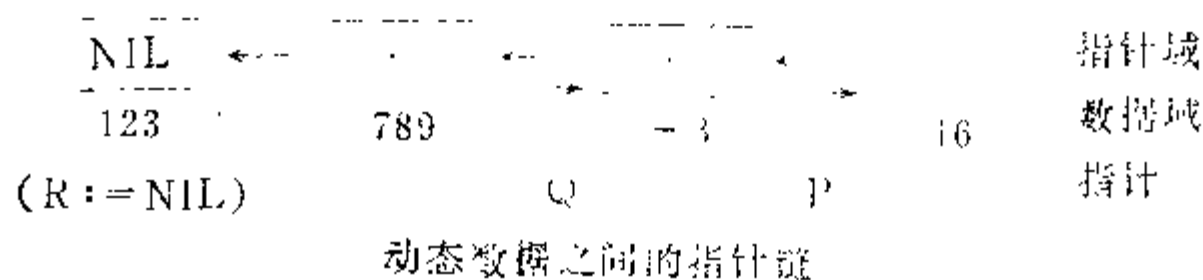
二、撤消一个动态数据

撤消一个动态数据，是通过DISPOSE过程完成的，如DISPOSE(P)。

DISPOSE(P)是NEW(P)的逆过程，它的作用是取消指针P当前所指向或称所引用的那个数据。因此，在调用这个过程之前，要先移动指针，使P指向要撤消的那个数据，可能还要再做另外一些处理。该过程执行完之后，这个数据就从用户程序中“消失”了，并释放它原来占据的内存区。至于对释放的内存区的管理或再次使用的问题，不同系统采用的办法不完全相同，可以“收集”起来，为后面的NEW过程重新使用，也可以不加处理，任其废掉。前者管理比较复杂，但可以节省程序运行过程中占用的内存空间。而后者不需要对这些内存区的收集与管理，程序运行可以更快些。

三、建立一批动态数据之间的联系

同一类型的一批动态数据之间，是用链表结构联系在一起的。解决它们之间的联系问题，实际上就是管好、用好指针链。让我们先看一个最简单的例子。



假定P和Q为指针，它们引用的数据由指针域和数据域两部分组成。指针域必须和P、Q同类型，并假定数据域为整型。上图就是由四个动态数据组成的链表结构。

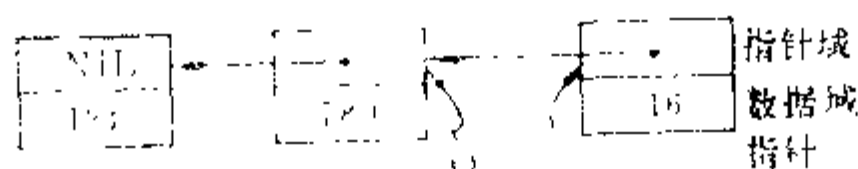
每调用一次NEW(P)过程，将建立一个新的数据区，P就指向这个数据区。如果后面还要通过调用NEW(P)过程建立下一个动态数据，则必须在新的一次NEW(P)调用之前，首先保存好当前P本身的值，否则新的NEW(P)一执行，原来保存在P中的内容(上一个动态数据的地址)将被新的P值取代，从而使前一次建立的动态数据变得

无法找到了，也就达不到建立一串动态数据的目的。为此，可以先用 $Q := P$ 这一赋值语句把原来 P 的内容赋给 Q ，在新的 $NEW(P)$ 调用之后，再用 $Q \uparrow \text{NEXT} := P$ 或 $P \uparrow \text{NEXT} := Q$ 的办法，把两个动态数据通过指针域勾链起来。反复多次执行上述操作，就能得到通过链表结构联系在一起的一串动态数据，如上图所示的样子。这里用的是 $P \uparrow \text{NEXT} = Q$ 的语句来建立链表的。

在撤消一个动态数据的操作过程中，同样要用到这个链表，并且，如果要撤消一串动态数据当中位置的某一个数据时，一定不能忘记首先把处在被撤消数据两侧的两个半截的数据串先勾链起来，否则可能造成丢失半串数据的错误。例如，要取消上图中 Q 所引用的那个数据，可用如下方式进行：

$P \uparrow \text{NEXT} := Q \uparrow \text{NEXT};$
 $\text{DISPOSE}(Q); \quad Q := P \uparrow \text{NEXT};$

操作后的结果如下图所示：



删除一个动态数据之后的情形

了解以上三点，就可以通过指针使用动态变量了。上面给出的是最简单的原理，实际上，指针类型可以比较容易地用于树形、双向链表或更为复杂的链表结构的数据管理。

4.6.3 应用指针的程序设计举例

下面让我们看几个应用指针与动态数据的程序设计的例子。

第一个程序，是建立一个简单的单向链表，用于记忆从终端键盘输入的数目不定的一些整数值。当输入的数值为 -1 时，结束输入，然后按与原输入次序相反的顺序输出已输入的全体数值。

其程序清单和运行结果如下：

```
PROGRAM PNT01A (INPUT, OUTPUT);
TYPE
  LINK = ^DATA;
  DATA = RECORD
    NEXT : LINK;
    INT   : INTEGER
  END;
VAR
  P, Q: LINK;
  I: INTEGER;
BEGIN
  (* PART 1 INPUT SOME INTEGERS *)
  (* AND MEMORY THEM IN DYNAMIC DATA *)
```

```

Q := NIL;
REPEAT
    READ (I);
    NEW (P);
    WITH P^ DO
        BEGIN
            NEXT := Q;    INT := I
        END;
    Q := P
UNTIL I = -1;
    (* PART 2 OUTPUT ALL INPUTTED INTEGERS *)
WHILE P <> NIL DO
    BEGIN
        WRITELN (P^.INT);
        P := P^.NEXT
    END
END.

```

INPUT:

32767 2345 5078 166 -15 -1

OUTPUT:

-1
-15
166
5078
2345
32767

在程序的类型说明部分，说明了一个指针类型标识符LINK和相应的一个数据类型标识符DATA。请注意这里对LINK和DATA说明次序，并注意DATA中有一个LINK类型的域，这是建立链表结构所必须的。

在变量说明部分，说明了两个LINK类型的指针变量P和Q。

在程序的执行部分，NEW(P)的作用只是分得需要的内存存储区，并把这片区域的地址送到P中，这片区域中的内容是通过另外两个赋值语句得到的。

为了建立和使用链表，一般要用两个或两个以上的相同类型的指针，Q开始赋为空值，用来记忆链表的一个端点。在NEW(P)之后，是用把Q的值赋给P^.NEXT，再把P的值赋给Q来建立链表的。

程序的第二部分用于显示已输入的数据值，程序中是用指针P走遍整个链，用Q也可以。把头一个数据的指针域赋为空，正好被用来控制结束引用链中数据的执行过程。

第二个程序与第一个程序的功能类似，只是把输出的顺序变为与输入相同的次序。

其程序清单和运行结果如下：

```

    PROGRAM PNT01B (INPUT, OUTPUT) ;
TYPE
    LINK = ^DATA;
    DATA = RECORD
        NEXT: LINK;
        INT : INTEGER
    END;
VAR
    P, Q, R:LINK;
    I : INTEGER;
BEGIN
    (* PART 1 INPUT SOME INTEGERS *)
    (* AND MEMORY THEM IN DYNAMIC DATA *)
    READ (I) ;
    NEW (P) ; R:=P; Q:=P;
    P^.INT:=I
    REPEAT
        READ (I) ;
        NEW (P) ;
        P^.INT:=I;
        Q^.NEXT := P;
        Q:=P
    UNTIL I = -1;
    P^.INT := -1;
    (* PART 2 OUTPUT ALL INPUTTED INTEGERS *)
    P := R;
    WHILE P<>NIL DO
        BEGIN
            Writeln (P^.INT) ;
            P := P^.NEXT
        END
    END.

```

INPUT:

32767 2345 5078 166 -15 -1

OUTPUT:

32767

2345

5078

166

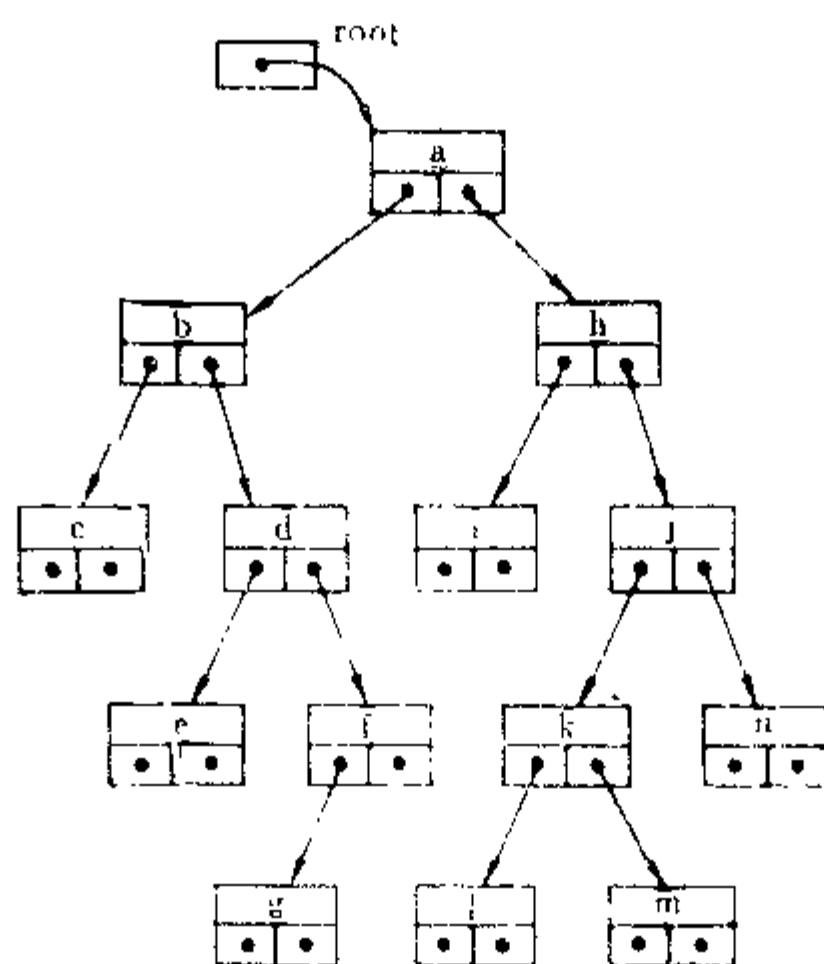
第三个程序，是用指针表示二叉树的一个例子。树本身是用递归方式定义的非常典型的一种数据结构，在许多应用领域中有重要使用价值，指针是表示和使用树形数据的良好工具。在本例子中，树的内容是以前缀次序的方式，记录在 TT.DAT 文件中。这个例子中用到了过程，是我们目前尚未学到的内容，但我们还是把这个例子放在本节，目的是让读者先大体了解使用指针类型能取得的效果。前两个程序中看不出使用指针的太多好处。对不大熟悉 PASCAL 语言中过程的说明和调用的读者，也可以在学完第五章的有关内容后，再读这个例子。

该程序的功能是以前缀、中缀和后缀的次序走遍整棵树。这里说的三种次序，是按用什么次序来处理根结点来划分的。若按从根结点开始，再左子树而后右子树的次序进行处理，就是前缀次序。若先左子树开始，再根结点而后右子树的次序进行处理，就是中缀次序。若先从左子树开始，再右子树而后根结点的次序处理，就是后缀次序。

TT.DAT文件的内容为:

abc..de..fg...hi..jkl..m..n..

这里每个小写英文字母代表树的一个结点的内容，每一个点“.”代表该结点的左子树或右子树指针为空（NIL）。这棵树的内容如下图所示：



在这个程序中，定义树的每个结点的数据为NODE类型，它由一项结点内容 INFO和左右两个指针LLINK、RLINK组成。程序中说明了四个过程，分别完成按前缀、中缀和后缀次序走遍整棵树，以及从 TT.DAT 文件中读来文件内容并建起树结构的数据内容的功能。程序的运行结果写到 TT1.DAT 文件中。

程序 1 编译和运行结果给出如下:

PROGRAM PNT02 (INPUT, OUTPUT) .

```

TYPE
  PTR = ^NODE;
  NODE = RECORD
    LLINK, RLINK: PTR;
    INFO : CHAR
  END;

VAR
  ROOT: PTR;
  CH: CHAR;
  F1, F: TEXT;

PROCEDURE PREORDER (P: PTR) ;
  BEGIN
    IF P < > NIL
    THEN
      BEGIN
        WRITE (F, P^.INFO) ;
        PREORDER (P^.LLINK) ;
        PREORDER (P^.RLINK)
      END
    END;

PROCEDURE INORDER (P: PTR) ;
  BEGIN
    IF P < > NIL
    THEN
      BEGIN
        INORDER (P^.LLINK) ;
        WRITE (F, P^.INFO) ;
        INORDER (P^.RLINK)
      END
    END;

PROCEDURE POSTORDER (P: PTR) ;
  BEGIN
    IF P < > NIL
    THEN
      BEGIN
        POSTORDER (P^.LLINK) ;
        POSTORDER (P^.RLINK) ;
        WRITE (F, P^.INFO)
      END
    END;

```

```

END,
PROCEDURE ENTER (VAR P:PTR) ,
BEGIN
  WRITE (CH) ,
  IF CH( )'.' THEN
    THEN
      BEGIN
        NEW (P) ;
        P^.INFO:=CH,
        ENTER (P^.LLINK) ;
        ENTER (P^.RLINK) ;
      END
    ELSE
      P:=NIL
  END,
BEGIN (*MAIN PROGRAM*)
  ASSIGN (F1,'TT.DAT') ; RESET (F1) ;
  ASSIGN (F,'TT1.DAT') ; REWRITE (F) ;
  ENTER (ROOT) ; WRITELN,
  PREORDER (ROOT) ; WRITELN (F) ;
  INORDER (ROOT) ; WRITELN (F) ;
  POSTORDER (ROOT) ; WRITELN (F) ;
  CLOSE (F)
END.

(*          TT.DAT          *)
(* abc..de..fg...hi..jkl..m..n..*)
(* OUTPUT: *)
(* abc..de..fg...hi..jkl..m..n..*)
(*          TT1.DAT          *)
(* abcdefghijklmn *)
(* cbedgfaihlknjn *)
(* cegfdbilmknjha *)

```

4.7 相同类型和兼容类型问题

MS PASCAL语言采用标准PASCAL语言的类型兼容规则。对扩充的数据类型，如高级数组类型、LSTRING类型和有关常量的处理上，又补充了某些新的兼容规则。类型变换功能是可用的。

相同类型、兼容类型或者不兼容类型是讨论两种类型之间的关系时必须区分的三种情况。

相同类型：有着同一标识符的两个类型，或者用TYPE T1=T2方式说明的两个等同类类型都属于相同类型。这里说的相同类型是真正的相同，两个相同的类型之间不存在任何区别，例如上面T1和T2就属于这种情况。

在需要的地方，某些常量可以变换类型。例如，整型常量可以变为字类型，字符类型常量可以变为STRING(1)类型，STRING(n)的常量可以变为LSTRING(n)类型等等。

如果要在两个记录之间，或两个数组之间进行赋值操作，则它们必须属于相同类型才能进行。但对字符串类型有些例外。

STRING(n)正好是PACKED ARRAY[1..n] OF CHAR的缩写形式，二者属相同类型。

正像前面已提到的，LSTRING(n)与PACKED ARRAY[0..n] OF CHAR不是相同的类型，也不属于兼容的类型。对于LSTRING类型的变量，在赋值、比较、读入和输出时都有一些特定处理。

兼容类型：两个简单类型的兼容条件是：

- 它们是相同类型；
- 一个是另一个的子界类型；
- 它们都是兼容型的子界类型；
- 它们都是ADR类型；
- 它们都是ADS类型。

两个构造类型的兼容条件是：

- 两个都不是文件类型，都不包含文件类型，并且都不是高级数组类型的构造类型所兼容的条件是，它们必须是相同类型；
- 两个拥有兼容基类型的集合类型的兼容条件是，它们都是PACKED（紧缩）集合类型，或者同为非紧缩的集合类型；
- 带有相同上界的STRING的派生类型相互兼容；
- 所有LSTRING的派生类型相互兼容。

了解两个数据（变量）类型的兼容性，是处理、检查表达式的正确性、赋值语句的合法性的的重要依据，是学习PASCAL语言程序设计必须掌握的内容。

习 题

1. PASCAL语言对数组的维数和下标的类型、下标的取值范围是怎么规定的？一个数组所包含的元素数目是在什么时间定义的？标准PASCAL语言不支持动态（变长）数组是什么意思？

2. 使用记录类型数据的好处表现在哪些方面？假若没有记录类型，要表明一批人的完整情况，会遇到些什么问题？

3. 记录中的变体部分主要用来解决什么问题？如果不设置变体部分，会给前面给出的例子中表示学生政治面貌情况带来什么困难？

4. 集合类型数据与数组类型数据、记录类型数据在用法上的差别表现在什么地方？能在集合数据上执行哪些操作运算？使用集合数据会给程序带来些什么好处？

5. 使用文件与使用其它三种构造类型数据的方法有什么不同？这种不同是怎么形成

的?说明程序中使用文件的必要性。

6. 写一个程序,完成简单的矩阵加法与乘法运算。

7. 对比然后修改本章第二节和第四节给出的两个程序STUDENTS和SALARY数据读入部分中的读入语句,体验并理解从终端读入整型变量、实型变量和字符型或字符数组型变量时输入数据的衔接关系,体会READ和READLN语句的合理选用问题。

8. 不用集合型数据,写出实现本章第三节中给出的程序SETS的功能的程序,并分析两个程序的清晰和易读性,比较两个程序的运行速度(如果PASCAL语言中有TIME函数,比较运行速度的事情简单些,否则略微复杂一点,准确性也会差一些)。

9. 写出对两个有序实型数据文件进行数据合并,从而得到一个新的有序实型数据的文件的程序。

10. 写一个实现对英文文章进行格式处理的程序,不管原来打字的格式是否正确,要求得到的文章格式满足下列要求:

(a) 每行最多80个字符;

(b) 每一英文句子头一个字母要大写,其它字母一律小写;一行中最后放不下一个完整的单词时,执行时要满足英文单词折行处理的正常规则;

(c) 任何一行不得以标点符号开头;

(d) 每一段文章的头一个单词要后退两个字符位置。

11. 在程序SALARY中加入统计男女人数的功能,按百元工资分档统计100元及100以下,101-200元,201元-300元,300元以上各档的人数的功能,并建立一个用工资额排序的新的顺序文件,再把这个程序进一步改成能对文件记录的任何域进行修改的程序。

12. 说明静态数据与动态数据在使用方法上的区别。

13. 说明指针本身的值与它指向的数据之间的关系,说明为什么指针指向的数据一般都是记录类型的?又为什么在该类型中至少要定义一个是指针本身类型的域?

14. 改写本章第六节给出的第一个程序,变原来的单向链表为双向链表关系,即链中间的每个数据都同时指向前后两个相邻的数据。

15. PASCAL语言不支持变长数组,你能用指针实现动态数组的功能吗?例如,读入一个正整数N的值,再把从1到这个正整数值之间的每一个整数值记入数组中,当N以从100-4096之间变化时,怎么处理才更好一些呢?此时完全可以先定义一个由100个整型值组成的一维数组,再定义相应的指针类型,然后根据读入的N,通次执行 $N \text{ DIV } 100$ 次的NEW操作的办法,得到一个动态的数组链,这时程序中数据区所用的内存单元,大体上取决于N,而不必先开一个4096个整型元素的静态数组。请你用一个程序实现这一思路。

16. 当用高级数组类型定义指针类型时,对NEW和DISPOSE过程的实际参数有什么特殊要求?

17. 高级数组类型的主要用途是什么?

18. 使用地址类型会给程序设计带来些什么好处?“危险性”又表现在什么地方?ADR和ADS的区别是什么?

19. STRING和LSTRING两种压缩字符数组,使用中的同异之处表现在哪些方面?

20. 地址类型和指针类型的区别表现在哪些方面?用这两种数据类型主要解决些什么方面的问题?

21. 何谓相同类型?何谓兼容类型?说明它们在分析表达式正确性方面的作用。

第五章 过程和函数

PASCAL语言是进行结构程序设计比较理想的语言之一，在程序结构、语句构造和数据类型等各方面都能很好地支持结构程序设计的有关规则。标准PASCAL语言支持程序设计中的模块和层次结构的基础是过程和函数，它是提高大型软件的设计质量、加快程序设计进度的关键技术。MS PASCAL语言还提供了MODULE和UNIT两种与程序结构类似的程序设计手段，从而能在更高层次上支持结构程序设计，就它们关键用法而言，MODULE和UNIT都是为其它程序提供可以选用的过程和函数的程序段。

过程和函数，都是PASCAL程序的一个组成部分，和汇编语言中的子程序，或软件中常说的例行程序等是一样的，只在被调用后才投入运行，并产生一定的处理结果，在它结束运行之后，则返回主调用者程序继续运行主调用者程序的后续语句。

从使用过程和函数的角度区分，可以把过程和函数分为两大类。一类是由PASCAL编译程序直接提供的，程序设计人员按语言说明书中给出的名字和规定的参数要求可以直接调用，如过去多次用到的READ、WRITE和SIN等过程和函数。我们通常称这一类过程和函数为预说明的（有时也说是标准的）过程和函数。另一类则要求由用户来说明和实现。这些过程和函数的名称叫什么，用什么类型的参数、完成哪些处理和得到什么运行结果（完成什么功能），都要由用户自己安排。我们通常称这一类过程和函数为用户自定义的过程和函数。

本章的第一节，将按过程和函数完成的功能，完整地分类介绍MS PASCAL语言的预说明的和可以使用过程和函数。接下来的各节将讲解用户定义及调用过程和函数的方法，过程和函数的参数类型，过程和函数的并列与嵌套结构，过程和函数的递归调用等有关概念。

过程和函数是有区别的。在说明时，所用的关键字有所不同，而且，在说明函数时还必须指明函数运算的结果类型。在使用时，过程调用本身是单独一个语句，而函数引用则必须出现在表达式中。

MS PASCAL提供的MODULE和UNIT的有关内容，已安排在本书第一章的第三节，本章内不再叙述，只在最后给出的两个简单的程序实例中用到了由MODULE和UNIT提供的过程和函数。

本章的内容比较复杂，然而又比较重要，要下一点功夫才能准确深入地掌握这些知识。概念要准确，用法要合理。使用过程和函数，是实现程序设计中模块与层次结构的基础，是提高PASCAL语言的大型软件的设计质量、加快程序设计进度的关键措施之一。

5.1 可用的过程和函数

MS PASCAL语言为用户提供了众多过程和函数。从用户使用的角度看，我们可以把这些过程和函数分为如下两大类：

预先说明的过程和函数。它们已在MS PASCAL的编译程序中被预先说明了，用

户可以在自己的程序中直接调用它们，不必做另外的说明。

· 标准的库过程和库函数。它们并未在MS PASCAL的编译程序中被预先说明，它们属于MS PASCAL标准运行库的一部分。用户若要使用它们，必须把它们说明为自己程序的外部过程和外部函数。

就预先说明的过程和函数来看，编译程序对它们采用了两种截然不同的处理方式。对绝大多数的过程和函数（例如：SIN，RESET等），确实在编译程序中被作为过程或函数说明了，用户在自己程序中使用它们，也真的是用调用过程或引用函数的方式实现的。另外一些过程和函数（例如：ORD，RETYPE等）则不然，编译程序将把它们转换成调用另外一段子程序，或直接把它们转换成特定的一段机器指令代码。

预先说明的过程和函数的名字，是PASCAL语言的标准标识符。按理说，用户是能够在自己的程序中把它们重新定义，或把它们转义使用的，但我们明确建议大家不要这样做。这既可以防止某些不必要的混淆，有利于改善程序的可读性，有利于交流，也有利于提高程序设计的质量。这一原则同样适用于对标准的库过程和库函数的使用方法上。

MS PASCAL给出了标准PASCAL语言提供的全部过程和函数，我们称它们为“标准”过程和函数。如不特殊说明，它们的参数是数值参数。此外，MS PASCAL还提供了另外一些“扩展”的过程和函数，这些扩展的功能的移植性较差。从程序的可移植性考虑，在准备移植到其它型号计算机系统的程序中，应避免使用这些扩展的过程和函数。

下面分类介绍MS PASCAL提供的全部过程和函数。

1. 动态分配内存的过程

这包括两个通过指针变量申请或释放一片内存区的过程NEW和DISPOSE。

PROCEDURE NEW (VAR P: 指针类型);

为指针变量P所引用的一个新变量分配所必须的一片内存区，并把这片内存区的首地址赋值给P。如果这个变量是带有变体的记录类型，将按变体可能用到的最大存储空间进行分配。

如果变量是高级数组类型，或是带有变体的记录类型，而且希望按所用变体成分实际需要的容量（而不是按变体最大需要）分配内存区，就必须使用长格式的NEW过程。

PROCEDURE NEW (VAR P: 指针类型; T1, T2, ... Tn: 高级数组下标的上界值);

PROCEDURE NEW (VAR P: 指针类型; T1, T2, ... Tn: 变体标志域的值);
这是NEW过程的两种长格式的用法。

当通过指针变量P用NEW过程来为高级数组类型的变量分配存储区时，必须在指针变量之后给出高级数组下标的上界值，以便正确地确定所分区域的大小。

通过这种办法得到的高级数组，一般不能执行对整个数组的赋值和比较，而是每次使用该数组的一个元素。只有LSTRING类型的高级数组是个例外，此时仍然能引用整个数组。如果过程和函数中用到了相同类型的高级数组的变量形式参数，用这种办法得到任何完整高级数组，都可以作为实参，整体地传送给调用的过程和函数。

当指针变量引用的是带有变体的记录类型的数据时，可以在指针变量之后，再给出本次的NEW过程中用到的变体成分的值，使内存分配能按该变体成分实际需要的内存区大

小来进行。这些变体标志域的值，必须按记录说明中的次序给出，而且任何一个后部的标志域都可以省略。

如果全部标志域的值均为常量，则按本次所需的实际存储区大小，在堆中进行分配。当所需容量为奇数个字节时，要再多分一个字节，凑成整字数目。如果哪一个标志域未明确给出用到的值，将按标志域所需的最大区域进行分配。

如果某些标志域的值不为常量值，编译程序将对首先遇到的非常量值的标志域，以及跟在它后面的每一个有未知值的标志域，都按它们所需要的最大存储区来进行分配。

在使用这样的动态变量时，程序员必须在NEW过程之后，为每个变体域进行正确的赋值操作，决不要再改变它们的值。编译程序不会对这些域赋值，也不检查这些域是否被正确地赋过值，也不去检查在程序的执行过程中这些域的值是否被修改过。

在标准PASCAL语言中，不允许把用长格式NEW过程建立的这种记录变量，作为表达式中的操作数，也不能把它作为过程或函数的参数传递，也不能向它赋值。MS PASCAL将不进行这种错误检查。正常的用法是，每次都使用该记录的各个域。

把一个大的记录赋给用NEW的长格式分配的较小记录，会冲掉堆中的部分内容。编译期间是很难检查出这类错误的。所以，在向堆中的带有变体的记录赋值时，要用该记录的实际长度，而不是记录的最大长度。

然而，向变体域进行不正确的赋值操作，很有可能要破坏堆中的某些变量，甚至会破坏堆结构本身。这类错误是检查不出来的。用户在使用中要特别当心。

PROCEDURE DISPOSE (VAR P:指针类型);

释放由指针P所指向的一片内存区。此时要求P必须是一个有效的指针，它不能为NIL值，也不能是没赋过值的、或指向堆中的已通过DISPOSE过程释放了的一个变量的指针（如果用了\$NILCK+，系统会检查这种错误）。

P也不应该是变量形式参数，或用在WITH语句中的记录指针，但这种错误是不进行检查的。

可以用这种短格式的DISPOSE过程安全地释放堆中的任何一个变量，不论P引用的变量是高级数组或带有变体的记录，也不论该变量是用长格式的或短格式的NEW过程分配的，都能保证正确的运行结果。请注意用这种短格式的DISPOSE，来释放用长格式的NEW在堆中分配的一个变量，是标准PASCAL语言中的一个难以查出的错误。

PROCEDURE DISPOSE (VAR P:指针类型; T1, T2, ...Tn:高级数组下标的上界值);

PROCEDURE DISPOSE (VAR P:指针类型; T1, T2, ...Tn:变体标志域的值);
这是DISPOSE过程的两种长格式的形式。

它同样用于释放由指针P指向的一片内存区，比短格式的DISPOSE过程多了个按T1, T2, ...Tn的规定对释放的变量内存区长度的检查。这里的T1, T2, ...Tn的定义和用法与NEW过程中的相同。

SIZEOF函数（随后将详细讲）在上述用法中是很有用的，我们可以用它取得带有相同上界的数组变量、或具有相同标志域值参数的记录变量要占用内存的字节数目。

2. 数据变换的过程和函数

这些过程和函数的基本功能，是把一种类型的数据变换成另外一种类型的数据。

FUNCTION TRUNC (Y:REAL):INTEGER

FUNCTION ROUND (Y:REAL):INTEGER

这是变实型数据为整型数据的两个函数。前者用舍掉自变量Y的小数部分的办法得到一个整数，后者则要对小数部分执行四舍五入处理。

例如，TRUNC (1.4) = 1, TRUNC (-3.9) = -3,

ROUND (1.4) = 1, ROUND (-3.9) = -4

如果求得整数值的绝对值大于MAXINT则出错。

FUNCTION FLOAT (X:INTEGER):REAL

这是变整型量为实型量的一个函数。通常情况下，程序员不必在自己的程序中进行这种变换，因为在需要将整型量变换为实型量时，这种变换会自动完成（由编译程序处理的）。运行软件包中要有这个函数，因此把它包括在MS PASCAL的标准函数中。

FUNCTION ORD (X:有序类型):INTEGER

FUNCTION WRD (X:有序类型):WORD

这是变一个有序类型量的数据为整型数据和字型数据的两个函数。X可以是任何有序类型的一个数据，ORD函数得到一个整型数据，WRD函数得到一个字型数据。ORD为ORDER的缩写，WRD为WORD的缩写，容易记忆。

对ORD函数来说，自变量X和返回值有如下关系：

当X为整型量时，返回值为X本身。

当X为字型量时，如果 $X \leq \text{MAXINT}$ ，返回X的值，如果 $X > \text{MAXINT}$ ，返回值为 $X - 2 * (\text{MAXINT} + 1)$ ，实际上是同样的16个二进制的位，只是返回时把X的最高位看成负号，而不再是32768这个数值。

当X为字符型时，返回它的ASCII编码值。

当X为枚举类型时，返回在类型定义里X的位置编号，枚举类型常量值编号从0而不是从1开始。

当X为指针类型时，返回一个整型值。

对WRD函数来说，自变量X和函数结果有如下关系：

当X为字型量时，返回值为X本身

当X为整型量时，如果 $X \geq 0$ ，返回值与X相同，如果 $X < 0$ ，返回值为 $X + \text{MAXINT} + 1$ ，实际上是同样的16个二进制位的值，但把原来X的符号位上“1”看成32768这个值了。

当X为字符类型时，返回它的ASCII编码值。

当X为枚举类型时，返回在类型定义中X的位置编号。

当X为指针类型时，返回一个字型的值。

FUNCTION CHR (X:有序类型):CHAR

这是变有序类型的一个数据为字符类型的一个函数。X可以为任何有序数据，结果得到一个字符类型的量。它是把X作为结果字符的ASCII编码来处理的。如果ORD(X) > 255并且\$RANGECK为选用状态，则出错。

在标准PASCAL语言中，规定X只能是整型量，MS PASCAL对它进行了扩展，规定X可以为任何一种有序类型的数据。

FUNCTION ODD (X:有序类型):BOOLEAN

这是用于判断任何一种有序类型的一个数值是否为奇数的函数，X为奇数时，返回值为真，否则返回值为假。

FUNCTION PRED (X:有序类型):有序类型

FUNCTION SUCC (X:有序类型):有序类型

这是求出任何有序类型的一个量X之前或之后的另一个量的两个函数。X可以是任何有序类型的数据，结果为同一类型的另一个数据值。对PRED函数来说，返回值为排在X之前的那个值，对SUCC函数来说，返回值为排在X之后的那个值，返回值与X的关系可以被表示成：

$ORD(\text{返回值}) = ORD(X) - 1$ 或

$ORD(\text{返回值}) = ORD(X) + 1$

如果返回值超出定义的范围并且\$RANGECK为启用状态，则可以检出这个错误。

如果X为整型量并且结果溢出，则\$MATHCH为启用状态，运行时也会检查出这个错误。

严格说来，这两个函数都没有进行数据类型变换，但习惯上都把它们划归到这类函数中来。

PROCEDURE PACK (CONST A:非紧缩数组; I:下标值; VAR Z:紧缩数组);

PROCEDURE UNPACK (CONST Z:紧缩数组; VAR A:非紧缩数组; I:下标值);

这是用来完成在紧缩数组与非紧缩数组之间传送数组元素的两个过程。

假定 A是 ARRAY[M..N] OF T

 Z是 PACKED ARRAY[U..V] OF T

则PACK过程完成的功能相当于：

FOR J:=U TO V DO Z[J]:=A[J-U+1]

就是把非紧缩数组A的从下标值I开始的各元素，依次传送到紧缩数组Z的每一个元素中去。

对UNPACK过程，其完成的功能相当于：

FOR J:=U TO V DO A[J-U+1]:=Z[J]

就是把紧缩数组Z中的每一个元素，依次传送到非紧缩数组A的从下标值I开始的各元素中去。

在调用这两个过程时，要注意如下三个问题：

(1) 两个数组A和Z的元素类型必须相同，两个数组用非紧缩和紧缩两种不同的方式保存元素数据。

(2) 两个过程中的参数I，是用于数组A的起始下标值。I和数组的下标的界值必须合理，就是说，在非紧缩数组A中，下标从I到N所拥有的元素数目必须大于或等于紧缩数组Z所拥有的元素数，否则就是出错情况。\$RANGECK为启用状态时，将执行数组下标的界值检查。

(3) 要注意两个过程中参数的顺序，二者的参数顺序是不相同的。这里的CONST和VAR表示用的都是变量类型的参数。CONST为前缀的实参在该过程中不能接受赋值操

作，而VAR为前缀的实参要接受赋值操作。I用的是数值参数。

3. 算术运算的函数

算术运算是实现数值计算的基础。全部的算术运算函数，都要使用实型数据、整型数据及与整型兼容的数据。ABS和SQR函数也用于计算字类型的数据。

处理实型量的全部函数，都要检查所用实型量的合法性。如果遇到非初始化的一个实型量，并且给出了特定出错条件检查，则出现这类错误时，将产生运行错误。

ABS和SQR两个函数在处理整型和字类型数据时，如果\$MATHCK处在启用状态，遇到出错情况将产生运行错误。如果\$MATHCK处在停用状态，则不会给出运行错误，但运行结果可能没道理。算术运算函数包括：

ABS(X) 求X的绝对值，X可以为整型，字型，长整型和实型量。

SQR(X) 求X的平方值，X可以为整型、字型和实型量。

SQRT(X) 求X的平方根值。结果一定为实型量。如果 $X < 0$ 则出错。

SIN(X) 计算X的正弦值，X用弧度表示，结果为实型量。

COS(X) 计算X的余弦值，X用弧度表示，结果为实型量。

ARCTAN(X) 计算X的反正切值，结果为实型量，是弧度值。

EXP(X) 计算以e为底的X的指数值，结果为实型量。

LN(X) 计算以e为底的X的对数值，结果为实型量。如果 $X \leq 0$ 则出错。

另外有一些函数，在其它语言中可能支持的，而PASCAL语言中却没有，但这亦很容易用如下办法实现，例如：

$\text{MAX}(X, Y) = X + (Y - X) * \text{ORD}(X < Y)$

$\text{MIN}(X, Y) = X + (Y - X) * \text{ORD}(X > Y)$

$\text{SIGN}(X) = \text{ORD}(X > 0) - \text{ORD}(X < 0)$

$\text{POWER}(X, Y) = \text{EXP}(Y * \text{LN}(X))$

($X > 0$ 时，计算的 Y^X)

这些都很容易用PASCAL语言的用户定义的函数实现，而且，可以用PURE属性和\$RUNTIME编译命令处理。例如：

{ \$RUNTIME+ }

FUNCTION POWER(X, B:REAL):REAL[PURE];

BEGIN

IF $A \leq 0$ THEN ABORT('No plus real', 24, 0);

POWER := EXP(B * LN(A))

END;

4. 处理实数的函数

运行程序库还提供了另外一些处理实型数据的函数，用户必须在自己的程序中把它们说明为外部函数后才能引用。

RSIRQQ(A:REAL, B:INTEGER)

计算 A^B 的值

RSRRQQ(A, B:REAL)

计算 A^B 的值，A必须大于或等于0

MINRQQ(A, B:REAL)

返回A和B中那个数值小的实型值

MAXRQQ (A, B:REAL)

返回A和B中那个数值大的实型值

AT2RQQ (A, B:REAL)

返回A/B的反正切值

ATNRQQ (A:REAL)

返回A的正切值

ASNRQQ (A:REAL)

返回A的反正弦值

ACSRQQ (A:REAL)

返回A的反余弦值

TNHRQQ (A:REAL)

返回A的双曲线正切值

SNHRQQ (A:REAL)

返回A的双曲线正弦值

CHSRQQ (A:REAL)

返回A的双曲线余弦值

LNDRQQ (A:REAL)

返回A的以10为底的对数值

ANNRQQ (A:REAL)

返回A的整数部分对应的实型值

AINRQQ (A:REAL)

返回A的按四舍五入处理小数部分以后的整数部分对应的实型值

DXPRQQ (A:REAL) :INTEGER

返回A的以10为底的指数的整数值。例如 DXPRQQ (A) 的结果为E, 则

ABS (X) 的值要 $\geq 10^{E-1}$ 并且要 $< 10^E$ 。

M10RQQ (A:REAL, I:INTEGER)

返回A乘上 10^I 的值

MP2RQQ (A:REAL, I:INTEGER)

返回A乘上 2^I 的值

5. 扩展的内部特性

MS PASCAL提供了如下一些内部扩展的过程和函数

PROCEDURE ABORT (CONST STRING, WORD, WORD)

这个过程以内部运行出错的同样方式停止程序的执行过程。参数 STRING (或 LSTRING) 是出错信息, 第一个WORD类型参数是出错码, 第二个WORD类型的参数由用户自行安排, 并且它也出现在停止执行时给出的状态字符段中。

这几个参数和有关机器状态的几个数据, 例如程序计数器的值、堆栈框区指针和堆栈指针的值, 以及执行此ABORT的语句在源程序中的位置 (如果 \$LINE或 \$ENTRY 处在启用状态), 在结束执行时, 都将提供给用户。

如果 \$RUNTIME处于启用状态, 则机器状态信息, 将是用带有 \$RUMTIME 命令

进行编译时，首次调用该过程或函数的相应位置的状态信息。

FUNCTION LOWER (表达式):值

当表达式为数组、集合、枚举、子界类型时，该函数将返回有关它们的下界的一个值，这个值此时分别为数组下标的下界值，集合可用的最小的元素值，枚举类型的第一个元素值和子界类型的下界值。必须指出，该函数的参数（写成表达式）用的只是它的类型，而不是它的值，函数的返回结果才是与相应类型有关的一个值。

FUNCTION UPPER (表达式):值

该函数与LOWER函数是类似的，只是它的返回值是与相应类型有关的上界的值。这个上界值通常是一个常量。当参数类型为高级数组类型时，该函数将返回该高级数组下标的实际的上界值。

FUNCTION LOBYTE (INTEGER或WORD):INTEGER或WORD

FUNCTION HIBYTE (INTEGER或WORD):INTEGER或WORD

这是取一个整型量或字型量的低位字节或高位字节值的两个函数，结果（值为0..255）的类型将与参数的类型相同。请注意，低位字节是一个字的第一个还是第二个字节，取决于所用的处理器。对8086处理器，低位字节是第 $i+1$ 个字节，高位字节为第 i 个字节。

FUNCTION BYWORD (one-byte, one-byte):WORD

这是把两个用字节表示的有序类型的数据合成一个字型数据的函数。返回的字的高位字节部分为第一个字节参数，低位字节部分为第二个字节参数。再次强调，字节的顺序由高、低位字节来表明的，而不是用它们在内存中地址的先后来确定。

PROCEDURE EVAL (表达式, 表达式, ..., 表达式)

这个过程用来对它的参数本身进行计算，和用语句对表达式求值一样。参数可以为任何类型并且数目不限。可以在函数执行体内用来对函数求值。

FUNCTION RESULT (函数标识符):值

该函数用来读取所用函数的当前值，它只能用在所用函数的执行体内部，或用在被所用函数嵌套着的过程与函数执行体的内部。

FUNCTION SIZEOF (变量):WORD

FUNCTION SIZEOF (变量, 标志域1, 标志域2, ..., 标志域 n):WORD

返回变量占用的内存的字节数，该字节数是用字型的量表示的。标志域的值或数组的上界值，要按在NEW和DISPOSE过程中所用的办法来设置。如果变量是带有变体的记录，第一种格式返回的是该变量可能用到的最大内存区的字节数。如果变量是高级数组类型，就必须使用第二种格式。

FUNCTION ENCODE (VAR LSTRING, X:M:N):BOOLEAN

FUNCTION DECODE (CONST STRING, VAR X:M:N):BOOLEAN

这是两个在整型量，字型量、实型量和变长字符串类型的量之间进行数据变换的函数。前者变数量值为相应的变长字符串，后者变属于字符串类型的一个量为相应的数值量。当这种转换能正确完成时，函数的返回值为真，若转换过程中出现错误，函数的返回值为假，此时，对以VAR为前缀的参数不产生任何影响。在使用这两个函数时，一定要检查函数的返回值，保证所用的值确实是正确转换后得到的结果。

对ENCODE函数，X必须属于整型，字型，枚举类型以及它们的子界类型，X也

可以是布尔类型、实数类型,或者地址类型的·R或·S分量。ENCODE函数和WRITE语句的执行情况全完相同,包括场宽M和N的用法(请参见MS PASCAL语言对WRITE过程的用法和扩展用法)。这个函数出错的情况是LSTRING太短,容不下变换得到的全部字符,此时函数的返回值为假。

对DECODE函数,它将变换STRING类型或者LSTRING类型的一个量为某一数值量的机器内部的表示形式。这个函数的执行过程和READ语句是全完相同的,包括场宽M和N的用法。X必须是在说明ENCODE函数时提到的几种数据类型之一。这个函数出错的情况是,字符串内含有非法字符,或者变换后得到的结果溢出或超过指定子界的上下界值,此时,函数返回的结果为假,X中的值无意义。

字符串开头或结尾的空格字符、跳格字符(TAB),在变换过程中都不起作用,剩下的其它字符必须是LSTRING的组成成分。正确的用法,要保证它们都是合法的字符。

6. 系统内部特性

系统内部特性给出了如下一些过程和函数:

RETYPE (类型标识符, 表达式)

这是通用的类型转义函数。函数将按给定的类型返回表达式的计算值。通常情况下,给定类型所代表的数据长度,和表达式结果的长度应该相同,但该函数并不对此进行检查。对构造类型的变量,RETYPE函数允许在变量后跟有分量选择符,如数组的下标,记录的域名,指针和地址引用符等等。

RETYPE 是一种“危险”的类型转义操作,它可能不按设想的意图执行。还有另外两种方法可以用来实现类型变换

(1) 我们可以先说明一个带有变体的记录,使变体部分包括每一个要用的类型,这时可以指定表达式为一种变体成分,并且用另一个变体成分取返回来的值。这在标准语言一级中就能实现,并且不会产生错误。

(2) 也可以先说明一个所要类型的地址变量,然后通过ADR把其它变量的地址赋给它。

这些方法稍有不同,并且显得有些古怪,要尽可能地避免使用它们。

下面三个过程(MOVEL, MOVER, FILLC),都用到了ADRMEM类型的数值参数,而全部的ADR类型都是彼此兼容的,因此任何变量或常量的ADR地址都可以用作三个过程的实际参数。使用这三个过程有一定的危险性,但在某些情况下是很好用的。

```
PROCEDURE MOVEL (S, D:ADRMEM; L:WORD)
```

```
PROCEDURE MOVER (S, D:ADRMEM; L:WORD)
```

这是两个用于把S串中的L个字符移到D串中的过程,前者是按从左至右的次序移字符,后者是按从右至左的次序移字符。这里不进行范围检查,不管\$RANGECK或\$INDEXCK设置为启用或不用状态都是如此。

例如:

```
TYPE S10=STRING (10);
```

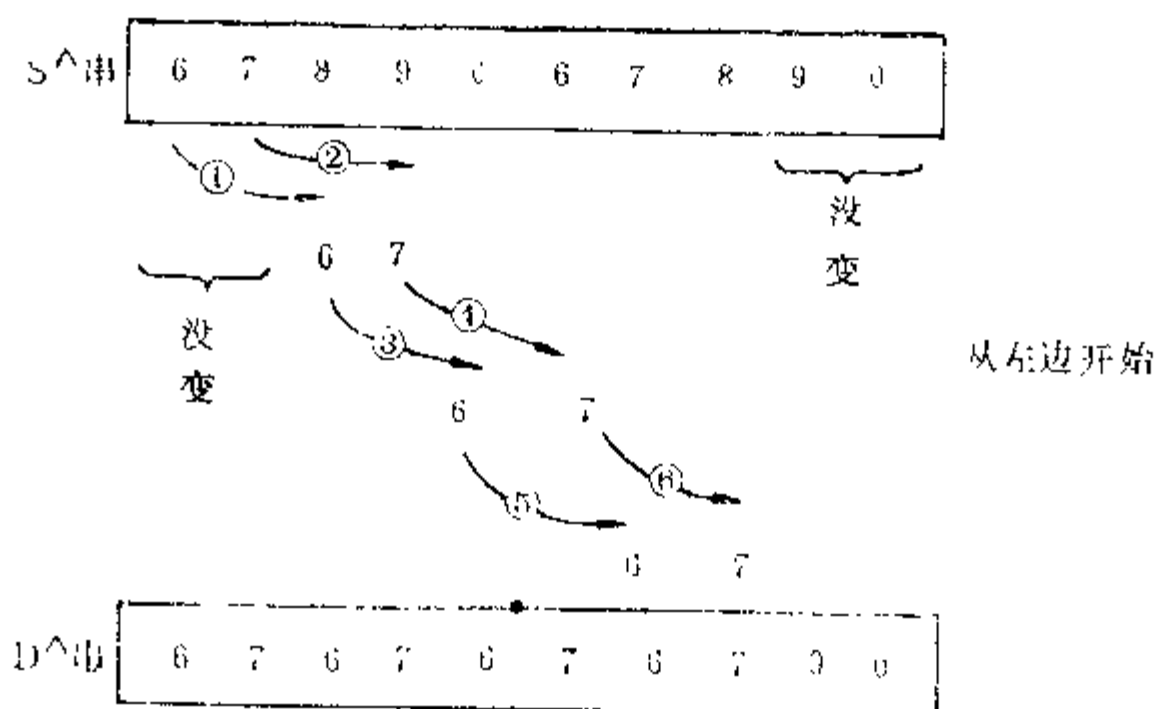
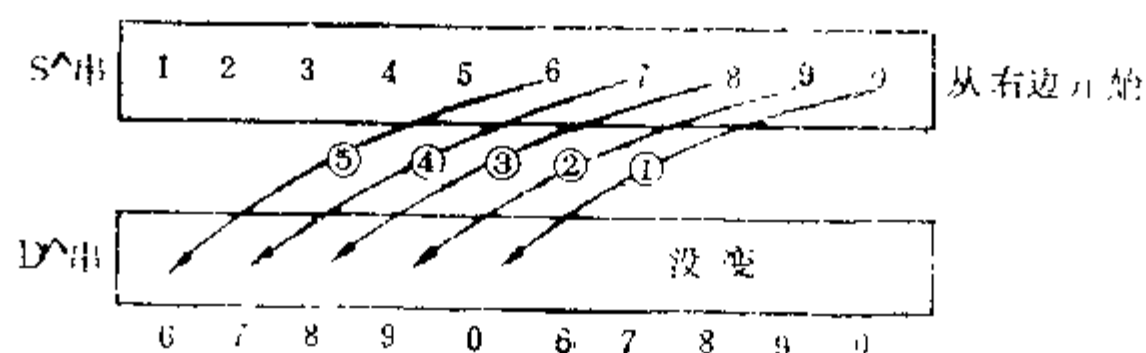
```
VAR ST:S10;
```

```

BEGIN
  ST:='1234567890',
  MOVER (ADR ST[ 6 ],ADR ST[ 1 ], 5),
  (* 结果为'6789067890' *)
  MOVEL (ADR ST[ 1 ], ADR ST[ 3 ], 6),
  (* 结果为'6767676790' *)
END.

```

我们可以把它们的执行情况表示如下:



PROCEDURE FILLC (D:ADRMEM, L:WORD, C:CHAR),

这个过程把字符C在D↑串从左向右依次写入L次, 与上两个过程一样, 这里也不进行边界值检查。

PROCEDURE FILLSC (D:ADSMEM, L:WORD, C:CHAR),

这个过程的功能与上一过程相同, 只是在参数部分用ADSMEM代替了ADRMEM, 实现段际地址串变量的填充操作。

与此类似的, 还有过程 MOVESL 和 MOVESR, 它也是用 ADSMEM 代替了 ADRMEN, 实现段际地址的串变量的传送操作。

必须说明, 当着把MOVE和FILL函数用于非紧缩的字符数组时 (即ARRAY [M..N] OF CHAR), 每个字符都要占用两个字节的位置, 一定要把它的每个元素都放置成全长度[]形式, 否则会产生某些非预期的错误。

7 字符串操作的内部特性

为了操作字符串, 编译程序给出了一些特定的过程和函数, 其中的某些过程和函数, 既

能用于STRING类型的数据，也能用于LSTRING类型的数据，有一些过程和函数则只能用于LSTRING类型的数据。

专用于LSTRING的过程

PROCEDURE CONCAT (VAR D:LSTRING, CONST S:STRING),

把S串接在D串的尾部，D串的长度变为原长度加上S串的长度。

如果UPPER(D) < LENGTH(D) + UPPER(S) 则出错。

PROCEDURE DELETE (VAR D:LSTRING, I, L:INTEGER)

在D串中，删除从D[I]开始的L个字符。D串的长度要减去L。

如果LENGTH(D) < I + L - 1 则出错。

PROCEDURE INSERT (CONST S:STRING, VAR D:LSTRING,
I:INTEGER)

把S串中的全部字符插入到D串中D[I]之前的位置，D串的长度要加上S串的长度。

如果UPPER(D) < UPPER(S) + LENGTH(D)，或者LENGTH(D) < I 则出错。

PROCEDURE COPYLST (CONST S:STRING, VAR D:LSTRING),

把S串的内容复制到D串中去，设置D串的长度为S串的长度。

如果UPPER(D) < UPPER(S) 则出错。

STRING和LSTRING合用的过程和函数

FUNCTION POSITN (CONST P:STRING, CONST S:STRING,
I:INTEGER) :INTEGER

返回模式串P在S串中的位置，查找是从S[I]位置开始的。如果找不到，或者I > UPPER(S)，返回值为0，如果P串中为空值，则返回1，没有出错条件。

FUNCTION SCANEQ (L:INTEGER, P:CHAR, CONST S:
STRING, I:INTEGER) :INTEGER

从S串的S[I]位置开始扫描，返回跳过的字符数。当遇到模式字符P，或已跳过L个字符，则停止扫描。如果L < 0，则扫描是从后向前进行，返回的也是负值。如果找不到模式字符P，返回值为L。如果I > UPPER(S)，则返回值为0。没有出错条件。

FUNCTION SCANNE (L:INTEGER, P:CHAR,
CONST S:STRING, I:INTEGER) :INTEGER

和SCANEQ功能类似，只是停止扫描的条件，仅为找不到模式字符P一项，与L的值不再有关。

PROCEDURE COPYSTR (CONST S:STRING; VAR D:STRING);

复制S串的内容到D串中。如果UPPER (D) <UPPER (S) 则出错。D串中剩余的字符用空白字符 (CHR (0)) 填充。

请注意，介绍系统内部特性时提到过的过程MOVEL、MOVER 和 FILLC 也是用于对串变量进行操作的有关过程。

8. 库过程和库函数

下列的过程和函数都没有被预先说明，用户要用它们，就必须先在自己的程序中用EXTERN命令来说明它们。

FUNCTION UADDOK (A, B:WORD; VAR C:WORD):BOOLEAN;

FUNCTION SADDOK (A, B:INTEGER; VAR C:INTEGER):
BOOLEAN;

FUNCTION UMULOK (A, B:WORD; VAR C:WORD):BOOLEAN;

FUNCTION SMULOK (A, B:INTEGER; VAR C:INTEGER):
BOOLEAN;

这是用来进行有符号的或无符号的16个二进制位的数算术运算的函数，在运算中产生溢出也不会给出运行错误。而正常的算术运算产生溢出一定导致运行错误，即使设置了\$MATHCK-也仍如此。上述四个函数，如果运算中不出现溢出，返回值为真，出现溢出，返回值为假。这四个函数，在实现扩充精度的算术运算中，或者在进行模 65536 的算术运算中，或者处理用户输入的数据的算术运算中，都是非常好用的。

FUNCTION ALLHQQ (SIZE:WORD):WORD

这是用于分配堆区的一个子程序。如果堆区已满，该函数返回 0 值，如果堆区结构有错，返回 1 值，或者返回指向已分配给变量的内存堆区的一个指针值。

PROCEDURE TIME (VAR S:STRING)

PROCEDURE DATE (VAR S:STRING)

这两个过程，将用“HH:MM:SS”或“DD:MM:YY”的格式，分别把系统中当前的时间或日期赋给STRING (或LSTRING) 类型的变量。它们是由DOS操作系统设置的，时间的格式是“小时:分钟:秒数”各由两位数字字符组成，日期的格式是“日:月:年”各由两位数字字符组成。

FUNCTION TICS:WORD

该函数返回DOS系统中定时器的时间值，以百分之一秒为单位，范围从 0 到99。定时器平均每秒变化18.2次，因此，返回值约以0.055秒的值递增。

PROCEDURE BEGXQQ

该过程全面开始运行例行程序，重新设置栈和堆区，初始化文件系统，调用BEGOQQ并调用程序体。这在程序运行过程中遇到严重暂时性错误，必须重新启动程序使其再次投入运行是有用的，所用文件均不关闭。

PROCEDURE ENDXQQ

该过程全面终止例行程序，调用ENDOQQ，终止文件系统，关闭已经打开的所有文件，并返回所用的操作系统（即不管调用的BEGXQQ）。它可用于在过程和函数内部不通过调用ABORT来结束程序的执行过程。

BEGOQQ和ENDOQQ 是分别在启动和结束一个程序期间被调用的。它们可以用于调用 DEBUGGER（调试程序），或写用户要用的诸如执行时间等各种信息。它们还可以用于其它的特殊目的。

FUNCTION DOSXQQ (COMMAND; BYTE; PARM; WORD); BYTE;
FUNCTION DOSXQQ (COMMAND, PARAMETER; WORD); BYTE;

前者是IBM PC机上的用法，后者是WANG PC机上的用法。这个函数是实现直接调用DOS操作系统的有关功能。函数中的第一个参数COMMAND将被传送到AH寄存器中，是BIOS的功能调用码，第二个参数PARAMETER将被传送到DX寄存器中。有关DOS功能调用的更详细内容，请参阅DOS的技术手册，下面我们只给出最常用的某些功能调用的具体用法。

(1) t1:=DOSXQQ (1,0)

把从键盘上打入的一个字符的ASCII编码值接收到字节变量t1中，并将该字符显示到终端屏幕上。如果执行此语句时，用户尚未敲键盘输入字符，该语句将等待输入。

(2) t1:=DOSXQQ (2,WRD ('X'))

把字符X显示到终端屏幕上，此时没有处理函数的返回值t1。该函数也可用于向终端传送CONTROL/S和CONTROL/Q命令，以实现屏幕内容卷动的（SCROLLING）停止和开始的控制功能。同样地，该函数也可以用于向终端发送CONTROL/P命令，以启动或停止让打印机打印终端屏幕内容的控制功能。

(3) t1:=DOSXQQ (6,255)

把刚打入的一个字符的ASCII编码值或一个零值接收到字节变量t1中。执行此语句时，当用户也从键盘打入了一个字符，t1中得到的是该字符的ASCII码值；若用户尚未入字符，则把一个零值送到t1中。打入的字符将不自动显示在屏幕上，而且对打入的CONTROL/S、CONTROL/Q和CONTROL/P不执行特定的功能处理。

(4) t1:=DOSXQQ (6,WRD ('X'))

把字符X显示到终端屏幕上，此时没有处理函数的返回值t1。这个语句的功能与前边的语句（1）是类似的，差别表现在它不对输出的CONTROL/S、CONTROL/Q和CONTROL/P执行特定的功能处理。

(5) t1:=DOSXQQ (11,0)

把终端工作状态信息取到了字节变量t1中，该语句不执行读字符的功能，仅用于了解终端运行状态。如果用户已从键盘打入了一个字符，t1中得到的值为255，尚未打入字符，t1中的返回值为0。

下面给出一个应当DOSXQQ函数的程序实例。

```

program dosxqq12(input,output);
var
  t:byte;   i,j: integer;
  function dosxqq(c: byte; parm: word):byte; extern;

begin
  for i:=1 to 10 do
    [t:=dosxqq(1,0);
     for j:=1 to 5 do writeln('1234567890');
     t:=dosxqq(2, wrd(t))];

  for i:=1 to 10 do
    [repeat t:=dosxqq(6,255)until t<>0;
     t:=dosxqq(6, wrd(t))];

  t:=dosxqq(11,0)   writeln(t:4);
  repeat t:=dosxqq(11,0)until t=255;  writeln(t:4)
end.

```

5.2 过程说明和过程调用

过程说明的一般结构如下,

```

-----PROCEDURE-----过程名----- ( -----形式参数表----- ) -----,
说明部分
-----BEGIN----- 语 句 -----END----- , -----□

```

其中,

PROCEDURE是过程说明的标志,它是PASCAL语言的保留关键字。每一个过程说明必须有一个标志PROCEDURE,不允许几个过程说明共用一个标志。

过程名由用户自己定义,但必须符合用户标识符的规定。过程调用时就用这个过程名。

形式参数表是过程内使用的形式变量,并用来与主程序进行数据通讯。

过程的形式参数的一般格式如下,

```

-----形式参数名----- : -----参数类型----- □

```

其中,

形式参数名由用户按照用户标识符的规定去定义;

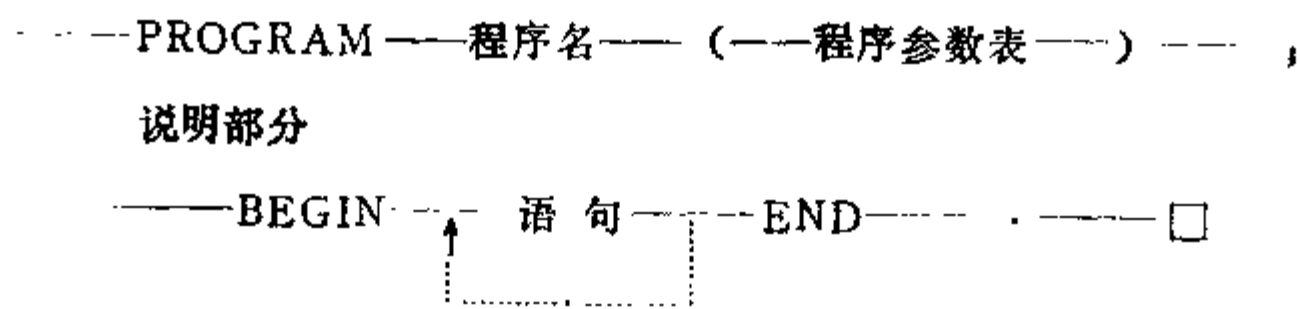
参数类型必须是标准的数据类型或已经定义的其它数据类型。

过程的说明部分与执行部分，和程序的说明部分与执行部分的意义大致相同，区别在于前者仅仅隶属于过程。过程可以有自己的标号说明，常量定义，类型定义，变量说明，还可以有自己的过程和函数的说明。

过程可以没有自己的参数表，甚至于没有自己的说明部分。

把过程说明的一般结构与PASCAL的程序的结构相比，会发现二者有许多类似之处，读者最好能看一看本书附录B中给出的语法图。我们这里给出了PASCAL程序结构的简化图，以便比较。但我们更要注意二者之间的区别，避免程序设计中的某些错误。

PASCAL语言的程序结构简化图如下：



程序结构与过程结构的区别在于：

第一，标志不同。程序的标志是保留关键字 PROGRAM，过程的标志是 PROCEDURE。

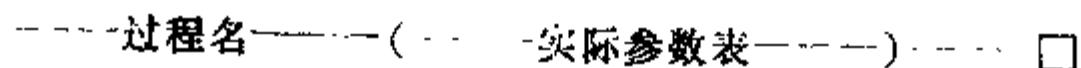
第二，参数不同。程序的参数表中给出的是某些类型的变量，如 INPUT, OUTPUT。它表示程序与外界之间联系的渠道。过程的形式参数表示过程与主程序之间的联系纽带。它可以是数值、变量、过程或函数。当然这些数值、变量或函数都必须属于某种数据类型。

第三，结尾不同。程序的结尾是英文句号“.”，表示程序到此总束。过程则是程序的一个组成部分，用分号“;”表示这个过程的结束。

第四，运行不同。程序的运行直接受操作系统控制，而过程的运行则由程序调用。程序可以调用过程，而过程是不能调用程序的。

过程调用，是通过在程序的执行部分给出过程名实现的。如果过程有自己的形式参数，在调用的过程名后面，还得在括号中给出本次调用的实际参数。过程调用本身是一个 PASCAL 语句。

过程调用语句一般格式如下：



其中：

过程名必须在程序的说明部分已加以说明，实际参数表对应于过程说明中的形式参数表。简单的过程语句可以没有实际参数。

过程调用语句的作用是，在程序执行遇到过程调用语句时，就转去执行相应的过程，即执行过程的执行部分，执行完后，返回到该过程调用语句的下一个语句继续执行主程序。过程语句中的实际参数为相应的过程提供某些“原始”数据。实际参数的个数与数据的类型、次序必须与形式参数表相匹配。这个问题还要在给出的程序例子中进一步说明。

5.3 函数说明和函数引用

函数说明:

函数说明与过程说明非常类似,其一般格式如下:

```

FUNCTION ---函数名---(---形式参数表---);---函数值类型---;
说明部分

      BEGIN ---语 句      END ---□
  
```

函数说明与过程说明的结构基本相同,主要有以下区别:

- (1) 函数说明的标志是关键字FUNCTION,过程说明的标志是PROCEDURE。
- (2) 过程说明允许没有形式参数表。但是,在一般情况下,函数说明必须有自己的形式参数,这些形式参数相当于函数的自变量。
- (3) 在函数说明的首部中必须指出函数值的数据类型。函数值的数据类型一般是标准数据类型,也可以是子界类型或指针类型等。
- (4) 在函数说明的执行部分至少必须有这样一个赋值语句,它将函数运算的结果赋给函数名,从而决定函数的值。

函数引用:

函数引用与过程调用不同,过程调用是一个独立的语句,函数不能作为一个单独的语句使用,函数只能出现在表达式中。这一点必须特别注意。函数名和它的参数在表达式中构成表达式的一个因子。函数引用是通过给出函数名和本次引用用到的实际参数实现的。函数必须是已在程序说明部分说明过的,用到的实际参数的个数、类型和次序必须与该函数形式参数表中的规定相匹配。

5.4 过程和函数的参数类型

在前面的讲解中,我们仅简单地未明确区别地给出和使用过程函数的两种类型的参数。实际上,在过程和函数中可以使用四种类型的参数,即数值型、变量型、过程型和函数型的参数。MS PASCAL语言的形式参数表完整的格式如下:

```

---CONST ---
---CONSTS ---
---VAR---
---VARs ---
形式变量名---;---类型---

--过程首部--
--函数首部--
  
```

其中:

形式变量名和类型的组成和用法与前面讲的变量说明非常类似。

VAR 是用在形式参数名前的一个标志, 是 PASCAL 语言的保留关键字, 表明它之后给出的参数为变量参数。如果形式参数为变量, 它前面又没有关键字 VAR, 则表示它为数值参数, 这就是我们在本节之前看到的情形。

CONST、CONSTS 和 VARS 用在形式变量名前面, 作为形式参数说明的前缀, 是 MS PASCAL 对标准 PASCAL 的一种扩展, 标准 PASCAL 是不使用这种前缀的, 它们代表的含义将在稍后的小节中进行说明。

过程、函数也可以作为过程和函数的参数使用。此时, 要在形式参数表中给出要用到的过程或函数的首部。在调用使用过程或函数型参数的过程时, 对应的实际参数必须是已经定义过的过程或函数。

这四种类型的参数的说明、用法和使用效果是不完全相同的, 我们分几小节逐一加以讲解。

5.4.1 数值参数和变量参数

从语法图上看, 数值参数和变量参数的说明是很相似的, 形式参数名都是变量名, 两者的差异主要表现为, 变量参数在说明时, 要在参数名前用保留关键字 VAR 作为前缀。

从使用的角度看, 这两类参数的用法与使用效果却明显不同。主要表现在:

(1) 在进行函数和过程调用时, 数值型实参可以是表达式, 变量型实参只能是变量;

(2) 在进行参数传递时, 对数值参数, 是先完成对实参表达式的计算, 把结果值传给被调用的过程或函数, 对变量参数, 是把实参在内存的地址传送给被调用的过程或函数;

(3) 在被调用的过程或函数运行期间, 这些过程或函数, 是用数值参数接收到的实参值直接运算, 而对变量参数, 则是用接收到的地址, 通过间址操作, 直接用实参本身进行各种运算;

(4) 从被调用过程或函数对主调用者的影响看, 使用数值参数, 不会对相应实参本身有任何影响, 但对变量参数则不然。如果在被调用的过程或函数内, 有向变量参数赋值的操作, 实参的值就被修改了, 这本是设置变量参数的意图之一, 即用变量参数接收过程处理的结果。然而, 如果在程序中处理不当, 这种修改实参本身的值的操作, 可能形成对调用者的一种“副作用”, 破坏了调用者本来的“原始数据”。后面给出的程序例子会对此进行解释。

(5) 当变量或数值参数属于构造类型数据时, 必须用已定义过的类型标识符来说明它们, 不能直接给出它们的定义, 这是保证形参与实参类型一致所要求的。当这样的参数本身是个数据个数很多的数据时, 例如一个大的数组, 把它说明为数值参数或变量参数, 对程序运行会有较大影响, 如果可能, 以把它说明为一个变量参数为好。第一, 节省内存, 因为对变量参数, 过程或函数的形参部分只为它保留一个地址单元就够了。对数值参数, 必须保留整个数组要占用的全部单元; 第二, 节省运行时间, 因为过程调用时, 对数值参

数，必须把实参的全部数值复制到形参中去，而变量参数只传一个地址。这样做的前提条件是，在被调用的过程运行期间，没有向数组赋值的操作，或者，实参被修改后对主调用程序的继续运行并无影响。

5.4.2 过程参数和函数参数

已经说明过的过程和函数，也可以作为其它过程和函数的参数使用。为此，必须在有关部分进行相应说明。

在形式参数表中，跟在关键保留字PROCEDURE之后的形式参数名，代表的是过程参数。跟在关键保留字FUNCTION之后的形式参数名，代表的是函数参数，对函数参数还必须给出函数参数的结果类型。

在调用带有过程参数、函数参数的过程或函数时，相应的实参必须是已经定义过的过程、函数。能作为实参使用的过程、函数本身还可能有自己的参数（对过程可以没有），一般规定，它们自己的参数只能是数值型参数。

标准PASCAL语言规定，标准过程和标准函数也可以做为实参使用，但有的PASCAL语言版本不支持这个功能，此时，用户要用标准过程和标准函数做为实参，就要在自己的程序中，先加一个辅助说明，例如，把标准过程SIN变为用户定义的过程SINE，办法如下：

```
FUNCTION    SINE (X, REAL); REAL;  
BEGIN  
    SINE:=SIN (X)  
END;
```

这之后，用户就可以在自己的程序中把SINE函数作为某些过程或函数的函数类型的参数使用。MS PASCAL语言中规定，可以把标准过程和标准函数作为过程或函数类型的参数使用。

我们准备对过程或函数类型的参数的各种问题做更多的说明，只给出两个简单的程序例子，说明使用它们的一般方法。

第一个程序，实现对 $\sqrt{1+X^2}$ 和SH(X)两个函数进行定积分运算。为此必须给出X值的下限和上限，以及计算过程中 ΔX 的增量值，或在上下限间的分段数。实现这一积分程序的清单如下：

```
PROGRAM INTEGRALS (INPUT, OUTPUT);  
CONST  
    N=100;  
VAR  
    X1, X2, AREA; REAL;  
    I; INTEGER;  
FUNCTION F1 (X; REAL); REAL;  
    BEGIN  
        F1:=SQRT (1+ X * SQR (X) )  
    END;
```

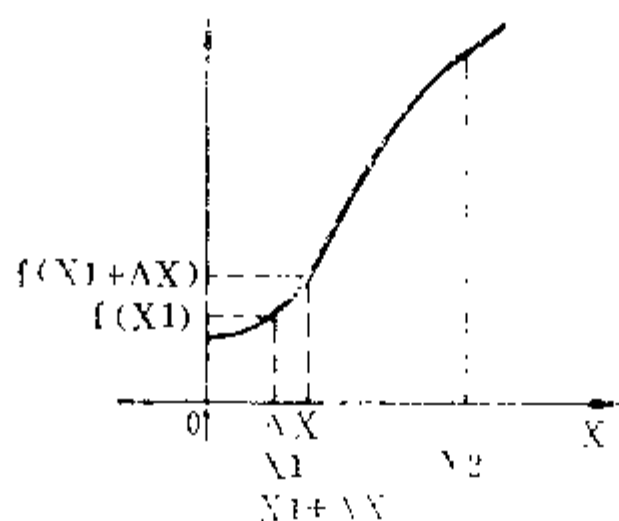
```

FUNCTION F2 (X:REAL) ;REAL,
  BEGIN
    F2:= (EXP (X) -EXP (-X) ) / 2
  END;
PROCEDURE INTEGRAL (VAR  AREA2:REAL, FUNCTION
  F:REAL) ,
  VAR
    Y1, Y2, X, AREA1, H:REAL,
  BEGIN
    H:= (X2-X1) /N, AREA1:=0;
    X:=X1, Y1:=F (X1) ,
    FOR I:=1 TO N DO
      BEGIN
        X:=X+H, Y2:=F (X) ,
        AREA1:= (Y1+Y2) *H/2,
        Y1:=Y2,
        AREA2:=AREA2+AREA1
      END
    END;
BEGIN (*MAIN PROGRAM PART*)
  WRITE ('ENTER THE LOW AND HIGH LIMIT OF X:'),
  READLN (X1, X2) ,
  IF X1<X2
    THEN
      BEGIN
        INTEGRAL (AREA, F1) ,
        WRITELN (AREA) ,
        INTEGRAL (AREA, F2) ,
        WRITELN (AREA)
      END .
    ELSE WRITELN ('INPUT ERROR! ')
  END.

```

这个程序中说明了两个函数和一个过程。两个函数对应我们要积分处理的函数，过程用于对给定的不同函数进行积分运算，它有个函数形参。主程序调用过程时，对应这个函数形参的实际参数就是上面说明的两个函数。从这里可以看到，用不同函数作为同一过程 INTEGRAL 的实参，就能实现对不同函数的积分运算。

用程序完成定积分运算，用的是逼近法。即把积分变量的变化曲线等效成若干折线，认为函数在每一份中的积分值为 $(f(X_1) + f(X_1 + \Delta X)) \cdot \Delta X / 2$ ，总的积分数为所有部分积分数之和。



计算的误差反映在 X 从 X_1 变得 $x_1 + \Delta X$ 时 $f(X)$ 不是线性变化的。当 ΔX 取得足够小时，还是能满足一般计算的精度要求的。

第二个程序用来模拟实验室中的恒温控制。这个控制与三个参数有关，即要求温度、实际温度和环境温度。实际温度比要求温度高，要送冷风降温；反过来，则要送热风升温，相同，不必调节。而环境温度比室内实际温度高，它会使室温上升，反之，会使室温下降。为了加快调节温度的过程，当实际温度与要求温度相差较大时，还应加大送风量，相差不大时，按需要正常送风。

我们假定，环境温度对室温的影响，按它与室温温差的约 $1/10$ 起作用。我们用一个过程ENVIRON来模拟它。

我们又假定，当实际温度与要求温度相差较大时，按单位时间内减少二者差的一半的速度进行调节，相差不大时，按单位时间内变化 2°C 的速度进行调节。我们用两个函数FSPEED和FNORMAL对应这两种办法。

又假定，要求恒温为 22°C ，正负变化范围各 1°C 。

要求从终端上输入当前室温与环境温度，计算并输出连续若干个单位时间的室温的情况。

程序清单如下：

```

PROGRAM TEMPERATURE (INPUT, OUTPUT);
CONST
  A = 22; B = 1;
VAR
  TEMP, TEMPOUTSIDE: INTEGER;
  I: INTEGER; F: TEXT;
PROCEDURE ENVIRON (X, Y: INTEGER);
BEGIN
  X := X + (Y - X) DIV 10;
  TEMP := X;
  WRITELN ('ENVIRON:', X:3)
END;
FUNCTION FSPEED (T: INTEGER): INTEGER;
VAR

```

```

        Y:INTEGER,
    BEGIN
        Y:=T+(A-T) DIV 2,
        FSPEED:=Y,
        WRITE ('FSPEED:', Y,3)
    END,
FUNCTION FNORMAL (T:INTEGER) :INTEGER,
    VAR
        Y:INTEGER,
    BEGIN
        IF T>=A+B THEN Y:=T-2,
        IF T<=A-B THEN Y:=T+2,
        FNORMAL:=Y,
        WRITE ('FNORMAL:', Y,3)
    END,
PROCEDURE REGULT (T:INTEGER, PROCEDURE P, FUNCTION F,
                    INTEGER) ,
    BEGIN
        WRITE ('REGULT', T,3) ,
        IF ABS (TEMP-A) >=B THEN T:=F (T)
        ELSE WRITE (' ', T,3) ,
        P (T, TEMPOUTSIDE) ,
    END,
BEGIN (*MAIN PROGRAM PART *)
    ASSIGN (F, 'LPT1:') ,
    REWRITE (F, ) ,
    WRITE ('ENTER TEMPERATURE INSIDE AND OUTSIDE ROOM:'),
    READLN (TEMP, TEMPOUTSIDE) ,
    WRITELN (F, 'TIMES: 1 2 3 4 5 6', ' 7 8 9 10 11 12 13 14 15') ,
    WRITE (F, 'TEMP.:') ,
    FOR I:=1 TO 15 DO
        BEGIN
            IF (TEMP>A+6) OR (TEMP<A-6)
                THEN REGUL (TEMP, ENVIRON, FSPEED)
                ELSE REGUL (TEMP, ENVIRON, FONRMAL) ,
            WRITE (F, TEMP, 4)
        END,
    WRITELN (F) ,
    CLOSE (F) ,
END.

```

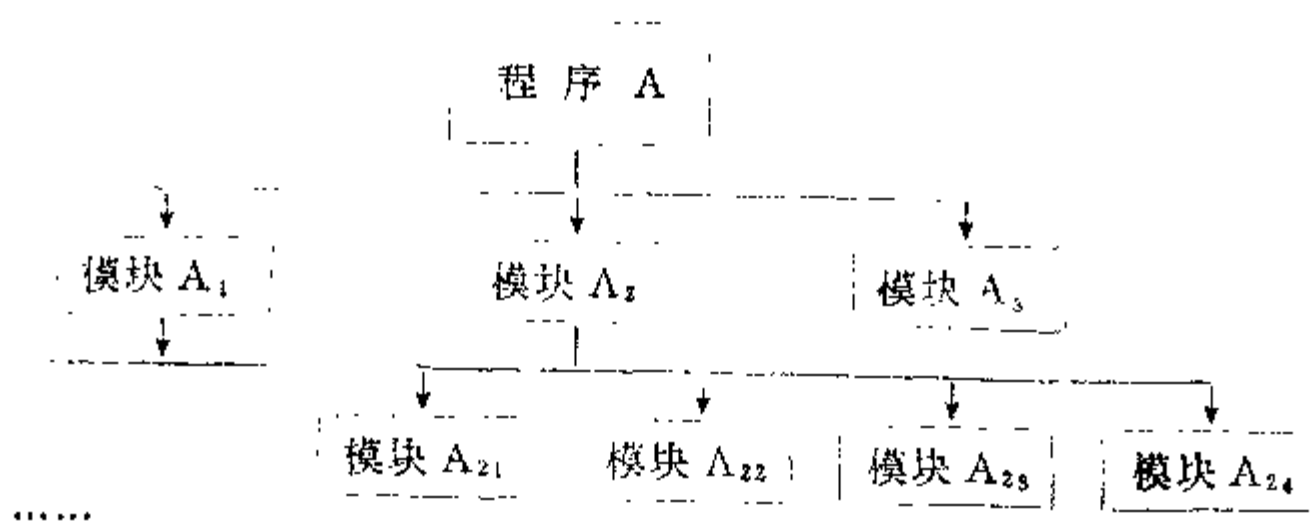
这个程序中，过程 REGULT 用了一个过程形参 P 去模拟环境对室温的影响，用了一个函数形参表达调节算法。从这个过程里的两个语句可以看出，当室温在所要求温度范围之内时，没有送风操作，但环境温度对室温的影响总是存在的，这个影响中所用的室温，是以送风后的计算值为起点的。

5.5 程序中的模块与层次结构设计问题

5.5.1 模块划分与模块设计

结构化程序中的一个核心思想之一，是实现程序设计中的模块化和层次化结构。所谓模块化，就是合理地把一个很长的程序，划分为若干个功能相对独立完整、彼此逻辑关系相对简单而接口联系尽量明确的程序块，并可以分头独立的设计与调试它们，在各自完成调试的基础上，再进行联调，最后完成整个程序的设计过程。这是对一个程序进行横向或称水平方向的划分过程。所谓层次化，是指对程序完成一次横向划分后，按实际需要，再对其中的一块或多块继续进行更细的划分，把它们划分成更小的模块。划分可以一直做下去，直到得到满意的结果。这种连续的逐步精细的划分过程，表现为程序的纵向或称为垂直方向的划分。用这种划分过程得到的程序模块之间存在着隶属和层次关系。在层次模块之间高层模块是低层模块的前提，要向低层模块提出功能要求；而低层模块是它的高层模块的功能的具体实现，它只受它的高层模块“约束”，只对它的高层软件“负责”，它们之间存在着严格的隶属关系。

下面是一个程序划分为模块的一个示意图。模块 A_1 ， A_2 ， A_3 是并列模块，都直接属于程序 A。



同理模块 A_{21} ， A_{22} ， A_{23} ， A_{24} 也是并列模块，它们都是模块 A_2 的下层模块。

结构化程序设计，不仅要求划分模块，而且还要求各模块在使用数据（如变量），彼此间调用的关系上满足一定的条件。例如，上图中模块 A_2 可以调用它的四个下层模块，但模块 A_1 和 A_3 不能调用它们。模块 A_{21} ，在需要时，可以使用模块 A_2 内部说明的数据，反过来则不行，等等。适当规定一些这样的限制条件，对简化程序设计和提高程序的质量是十分有益的。

怎样进行结构化程序设计，怎样划分程序模块，怎样规定层次模块间各种联系与限制条件等，都超出了本书讨论范围。但这里有两点推断，还是要讲清的：

第一，PASCAL 语言中的过程和函数，是实现结构化程序设计的重要手段，是实现

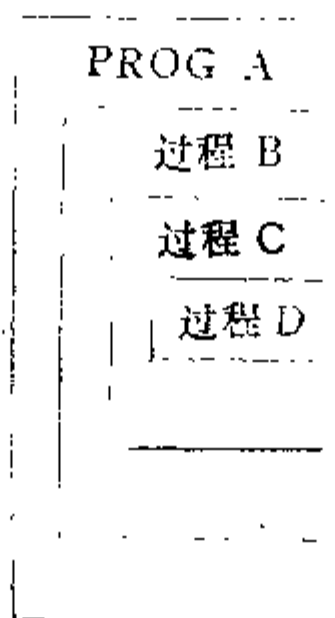
程序设计中模块化与层次化结构的理想工具；

第二，在学习过程和函数用法的过程中，有意识地把它与结构化程序设计的方法联系起来，对提高学习质量，加深对使用过程和函数的种种要求与限制的实质性的认识，是大有好处的。

下面我们分几小节，介绍在使用过程和函数的时候，与程序中的模块化，层次化结构有关的几个问题。

5.5.2 过程的嵌套和递归调用

从 PASCAL 的语法图上可以看到，过程和函数的说明是可以嵌套进行的。就是说，PASCAL 程序可以说明自己的一个或多个过程、函数，这些过程、函数又可以说明自己的过程和函数，并可以继续进行嵌套说明。例如，程序A说明了过程B，过程B又说明了过程C，过程C还可以说明过程D，这可以示意表示如下：



这是一种嵌套关系。即一个套着一个。这个嵌套的结构，反映的是程序模块中的层次关系。我们称套着别的一个过程的叫外层过程，被套的称为内层过程。例如，过程 D 是 C 的内层过程，C 又是 B 的内层过程，但它又是 D 的外层过程，B 也是 D 的外层过程。总共有多少层（不是多少个）过程逐层套在一起，称为过程的嵌套深度。这个嵌套深度是有个合理的限度的，标准的和大部分的 PASCAL 编译程序规定这个限度为七。我们经常把这里的 B 称为一层过程，C 称为二层过程等等。

嵌套的过程间，在使用数据和彼此调用的关系上，必须遵循下列规则：

1. 在数据使用上

在数据使用上，PASCAL 语言规定，内层过程可以使用自己说明的全部变量。如果用到那些不是自己说明的变量时，就要逐层到外层过程查找，可以一直找到主程序一级。这里的逐层是个重要的概念，就是说，只能在内层找不到时，才再往外找一层。而任何一个外层过程，一定不能使用它的内层过程说明的变量。因此，可以得出结论，PASCAL 语言的主程序，只能使用自己说明的变量。而任何一个过程，只要它自己和它的全部外层过程不使用与主程序中相同的变量名说明变量，它就一定有权使用主程序中说明的变量。我们通常称在主程序中说明的变量为全局变量（Global Variables），意思是说，如果需要，可以在整个程序中的任何地方使用它们。与此相对应的，在一个过程内说明的变量，只能在这个过程内部使用，称它们为这个过程的局部变量（Local Variables）。

这里说的只是变量，其实编译程序在编译用户源程序时，对常量名，类型名，过程与

函数名等进行处理时，也遵从了这一规则。

对不存在嵌套关系的过程来讲，它们的局部变量之间不存在任何联系。

2. 在彼此调用的关系上

嵌套的过程之间，在彼此的调用关系上，只允许外层过程（包括主程序）去调用自己的直接内层过程（由自己直接说明的，与自己层号差一），不允许隔层向内层调用。但是却允许内层过程直接调用自己的任何一层外层过程（当然不包括主程序，任何过程都不能调用主程序）。

也允许一个过程直接调用自己。我们通常称这种调用为过程的直接递归调用，这在解决一些本来是递归定义的问题时，十分有效。还有间接递归的例子。如B调用C，C调用D，D又调用B，此时，对过程B来说，它经过C和D又调用了自己。在写带有递归处理的（直接或间接的）过程时，要特别当心给出结束递归处理的判别条件。

在用过程实现程序设计中模块的层次结构时，必须充分注意到嵌套过程间在使用数据和彼此调用时的关系和条件。

5.5.3 并列过程间的调用关系

由同一过程（包括主程序）直接说明的多个过程为并列过程。并列过程有相同的层号。但具有相同层号，不直接隶属于同一个外层过程的那些过程间，不存在并列关系。

并列过程，是相对于嵌套过程而言的，它是对一个程序模块在“水平”方向上的划分结果。并列过程之间，彼此不能访问对方的局部变量。

在调用关系上，后说明的过程可以直接调用先说明的过程，因为这满足PASCAL语言对一个符号先说明后使用的要求。反过来，如果先说明的过程，还要求调用后说明的过程，就会与这一要求有矛盾。为解决这个问题，PASCAL语言中，又规定了一个“超前引用”说明。其具体办法是：在真正对后说明的过程进行说明之前，首先在先说明的过程之前，给出要后说明的那个过程的首部，并用“FORWORD”一词标明这是一个“超前引用”说明，使先说明的过程可以调用它，而它的真正说明，要到后面进行。

超前引用说明，是解决二并列过程之间相互调用所必须的。

应该注意到，并列过程的相互调用，也形成了过程的间接递归调用关系，必须处理好结束这种递归调用的条件。

在不存在并列与嵌套关系的过程之间，不能相互调用。但有一点必须说明，在并列过程之间，后说明的过程的内层过程，不仅能调用自己的外层过程，而且也能调用自己外层过程的那些已经说明了的并列过程。例如，主程序先说明了过程A，又说明了过程B，B又说明了自己的内层过程C。则B能调用A；C能调用B，也能调用A。但若不对B进行超前引用说明，A则不能调用B。A如果还有自己的内层过程，这些过程同样不能调用B。如果对过程B进行了超前引用说明，则刚刚说的不能调用的两种情况，都能进行调用了。但在A的内层过程与B的内层过程之间，无论如何也不能直接调用（但通过A或B可以实现间接调用）。这里说的是PASCAL语言允许实现这些调用，实践中应大力简化这些调用关系。

5.5.4 使用过程与降低内存占用量问题

在PASCAL程序中，正确合理地使用过程，可以减少程序运行期间占用的内存量。这是因为PASCAL程序中的全程变量，在程序投入运行的同时，就要为它们分配内存单

元，一直要占用下去，直到程序运行结束。而过程的局部变量，只是在这个过程被调用的期间为其分配内存单元，过程一结束，这些单元就可再分给别的过程用。过程总是轮流运行的，它们的局部变量不是同时占用内存。仅从这一点考虑，能用局部变量的地方，就不要用全程变量，此外，局部变量只限于过程内部使用，不受程序其它部分影响，用错的可能性要小些，更有利于提高程序的质量。

此外，从程序本身代码占用内存区的角度讲，使用过程和函数也是有益。

有的PASCAL的编译程序，支持在PASCAL程序中使用外部过程和外部函数，当连接程序支持复盖功能时，不同的外部过程和函数有可能共用同一片内存区，从而使整个程序比不用复盖技术时占用更少的内存。也有的PASCAL编译程序，支持对直接在程序中说明的过程、函数指明复盖结构。对这些内容，我们不在这里进行更多的讨论，我们将重点讲解MS PASCAL程序与内存占用量有关的几个问题。

MS PASCAL语言支持三种可编译的程序部，即程序、模块和单元。受PC机寻址机构的制约，每一个独立的MS PASCAL可编译部分最多能产生64KB的代码，这可能要有几千个PASCAL语句。当一个程序非常大，其代码在64KB以上时，就必须把该程序划分成一个程序和一或多个模块，或者一或多个单元的形式，经过连接之后，程序的代码就可以在64KB以上，可以达到二三百KB的规模。此外，还可以再有一个64KB的数据段。通过使用地址类型变量，还可以开辟另外的数据段。因此，一个MS PASCAL的程序能使用的最大内存容量不是64KB，也不是128KB，它往往是由PC机可用内存容量，或连接程序所能支持的程度来决定的。

MS PASCAL编译程序对一个程序能支持的标识符总数约为1000个。一般的程序是很难达到这个限度的。但在设计程序时，还是应合理地确定标识符。首先，以把标识符（在可能情况下）选得短一点为好，形象化些（与其代表的含义尽量接近）为好，此外，对UNIT接口中提供的某些标识符，有个重复计数这些标识符个数问题。例如，程序中用到一个接口A，接口A又用到一个接口B，则B中的某些标识符将被重复三次，A中的某些标识符将被重复两次。从这个角度看，一个程序中用到的标识符的个数可能是相当多的，有时可能不得不设法减少标识符的个数。

有关程序、模块和单元的特性及联系，我们已在本书第一章详细介绍了，还给出了几个应用实例。如果在读第一章时尚未完全读懂，现在完全可以读懂了。化一点精力读懂那里的程序实例，读懂本章最后给出的有关程序实例，对加深理解，MODULE和UNIT能在更高层次上支持结构程序设计的含义是十分必要的。

下面是应用高级数组指针和地址类型的一个程序例子。

```
program fileng(input,output);
type
  rec=record
    i2:integer;
    r4:real4;
    r8:real8;
    str:string(10)
  end;
var
```

```

prt: string;          iadr: adr of integer;
r4adr: adr of real4;   r8adr: adr of real8;
w: wofd;              f: file of rec;
i: integer;   r: real; str1: string(10);
sadr: adr of string(512);
s10adr: adr of string(10);

```

```

begin
  w:= sizeof(f^);          new(Prt,w);
  sadr:= adr prt^;         iadr.r:=sadr.r;
  r4adr.r:=iadr.r+2;       r8adr.r:=r4adr.r+4;
  s10adr.r:=r8adr.r+8;     assign(f,'filedata.dat');
  ,
  rewrite(f);
  for i:=1 to 10 do
    [f.^i2:=i;             f.r4:= i * 123;
     f.^r8:=i * 123/456;   f.str:='a123456789';put(f) ];
  close(f);

  reset(f);
  for i:=1 to 10 do
    [ move1(adr f,  adr prt,w);
      writeln(iadr:4,r4adr :15:8,r8adr^:20:15,s10adr^:14);
      if i<10 then get(f) ];
    close(f)
  end,

```

5.6 程序设计举例

第一程序用于求出从终端读入的一个正整数值的因子积的表示形式，用了一个简单过程 FACTOR。求因子用的是判断该整数值能否被 2 到 $\text{TRUNC}(\text{SQRT}(X))$ 之间数值整除，能整除，就表明找到了一个因子。这里要说明的是对试算的除数都用 WHILE 语句控制，以便求出连续的几个相等的因子。一个数的最大因子永远不会超过它的平方根值。如输入一个 60，则输出应为：2 * 2 * 3 * 5 * 1 = 60。

```

PROGRAM FACTORS (INPUT, OUTPUT),
VAR I, J, X, HIGHLIMIT, INTEGER,
    OK, BOOLEAN,
PROCEDURE FACTOR,
BEGIN
  HIGHLIMIT:=TRUNC (SQRT (X) );

```

```

    FOR I:= 2 TO HIGHLIMIT DO
      WHILE X DIV I=X/I DO
        BEGIN
          OK:=TRUE;
          X:=X DIV I;
          WRITE (I:4, ' * ')
        END
      END;
    BEGIN (* MAIN PROGRAM *)
      READ (X);      J:=X;
      WHILE X>= 6 DO
        BEGIN OK:=FALSE;
          FACTOR;
          IF OK
            THEN
              WRITELN (X:4, '=', J:5)
            ELSE WRITELN ('NO PROPER FACTOR');
          END;
        WRITELN ('PRPGRAM COMPLETED! ')
      END;
    (* 60
      2 * 2 * 3 * 5 * 1 =60
    PROGRAM COMPLETED! *)

```

第二个程序用于判断从终端读入的两个正整数X和Y是否为亲密数对。如果X的全部因子之和等于Y，而Y的全部因子之和等于X，则说它们是亲密数对。这里说的因子和是指能整除X的全部正整数之和，包括1但不包括X本身。如6的因子和为 $1 + 2 + 3$ ，等于6，而12的因子和为 $1 + 2 + 3 + 4 + 6 = 16$ 。

这个程序中用了一个带有两个形式参数和有两个局部变量的过程FACTOR，它实现的是计算一个正整数的因子和。主程序用X、X1和Y、Y1两组实际参数两次调用它，然后进行判断并输出相应结果。

```

    PROGRAM AMICABLE1 (INPUT, OUTPUT);
  LABEL
    1;
  VAR
    X, X1, Y, Y1:INTEGER;
    PROCEDURE FACTOR (J:INTEGER; VAR K:INTEGER);
    VAR
      I, LIMIT:INTEGER;
    BEGIN

```

```

        K:=1,
        LIMIT:=J DIV 2,
        FOR I:=2 TO LIMIT DO
            IF J/I=J DIV I THEN K:=K+1
        END,
BEGIN (*MAIN PART OF PROGRAM*)
    READLN (X, Y);
    IF (X>=6) AND (Y>=6)
    THEN
        BEGIN
            FACTOR (X, X1);
            FACTOR (Y, Y1);
            WRITELN (X1, Y1);
            IF (X=Y1) AND (Y=X1)
            THEN
                BEGIN
                    WRITELN ('These are two amicable numbers');
                    GOTO 1
                END
            END,
            WRITELN ('PROGRAM COMPLETED');
1:
END.
(* 220 284

```

```

                220                284
These are two amicable numbers    *)

```

这个程序中，过程的第二个形式参数一定要用变量参数，否则得不到正确的运行结果。请读者一定要注意并理解这个问题。

第三个程序的功能与第二个程序完全一样，但改用函数FACTOR 取代了原来的过程。这一变化对主程序的外貌有直接影响。因为使用过程要用单独一个过程调用语句，过程执行后要传给主程序的计算结果一般用变量参数完成，而函数调用却只能出现在表达式中，而且函数的执行结果用函数名传给调用者。

```

PROGRAM AMICABLE2 (INPUT, OUTPUT);
VAR X, Y: INTEGER;
FUNCTION FACTOR (J: INTEGER): INTEGER;
VAR I, K, LIMIT: INTEGER;
BEGIN
    K:=1;    LIMIT:=TRUNC (SQRT (J));
    FOR I:=2 TO LIMIT DO

```

```

        IF J/I=J DIV I
          THEN K:=K+1,
        FACTOR:=K.
      END,
BEGIN
  READ (X, Y) ,
  IF (FACTOR (Y) =X) AND (FACTOR (X) =Y)
    THEN WRITELN ('these are two amicable numbers')
    ELSE WRITELN ('PROGRAM COMPLETED')
  END.
  (* 12 16
  PROGRAM COMPLETED
  220 284
  these are two amicable numbers *)

```

第四个程序为把一个函数作为另一个过程的形式参数的例子。程序中先说明了一个函数FUNC,在说明过程PROC时用了个函数形式参数F0,并在过程的执行体部分调用F0。在主程序调用过程PROC时,用了个变量X和函数FUNC做为实际参数,就把主程序,过程PROC和函数FUNC全联系在一起了。

```

  PROGRAM FUNCPROC (OUTPUT) ,
VAR X, REAL,
  FUNCTION FUNC (Y:REAL) ,REAL,
  BEGIN
    FUNC:=SQR (Y) -2 * Y+2
  END,
  PROCEDURE PROC (VAR Z, REAL,
                  FUNCTION F0 (XX, REAL) , REAL) ;
  BEGIN
    Z:=F0 (Z)
  END,
BEGIN (* MAIN PROGRAM *)
  X:= 5,
  WRITELN (' X0=', X:7:2) ,
  PROC (X, FUNC) ,
  WRITELN (' X1=', X:7:2) ,
  PROC (X, FUNC) ,
  WRITELN (' X2=', X:7:2)
END.
(*
  X0=  5.00

```

```

X1= 17.00
X2=257.00 * )

```

第五个程序的功能与第四个完全相同，但没有把函数作为过程的形式参数使用，而是在过程中直接调用函数。给出这个程序的目的是让读者更直接地理解函数作为形参的意义和效能。在这种最简单的情况下，看不出使用函数参数的好处。如果主程序中说明了两个不同的函数，而只能在调用过程PROC时通过不同的函数实际参数使用不同的函数，在过程说明中使用函数形式参数就会给程序设计带来某些方便。

```

PROGRAM FUNCPROC (OUTPUT);
VAR X:REAL;
FUNCTION FUNC (Y:REAL):REAL;
BEGIN
    FUNC:=SQR (Y) -2 * Y+2
END;
PROCEDURE PROC (VAR Z:REAL);
BEGIN
    Z:=FUNC (Z)
END;
BEGIN (* MAIN PROGRAM *)
    X:=5.46;
    WRITELN (' X0=', X:7:2);
    PROC (X);
    WRITELN (' X1=', X:7:2);
    PROC (X);
    WRITELN (' X2=', X:7:2)
END.

(* X0=  5.46
   X1= 20.89
   X2=396.68   *)

```

第六个程序是函数使用过程形式参数的例子。必须说明，不同的PASCAL 语言版本对使用过程和函数形式参数的支持程度，使用中某些有关规定可能不完全相同，请阅读有关手册后再使用。但一般都规定，作为参数使用的过程和函数本身只能使用数值参数，以避免调用它们时，它们对主调用者的参数本身数值产生的“副”作用，也可以简化程序设计的难度。

```

PROGRAM PROFUN (INPUT, OUTPUT);
VAR
    I, J: INTEGER;
PROCEDURE P (X: INTEGER);
BEGIN

```

```

        X:=X*2,    J:=X
    END,
    FUNCTION F (VAR Y, INTEGER,
                PROCEDURE PRO (X, INTEGER) ), INTEGER,
    BEGIN
        PRO (Y) ,    F:=J+100
    END,
BEGIN
    J:=3,    J:=F (I, P) ,
    WRITELN ('I=', I, 4, '    J=', J, 4)
END.
(*    I= 3    J=106    *)

```

第七个程序是应用过程嵌套的例子。请注意它们之间变量的使用关系和调用关系。

```

PROGRAM NESTED (OUTPUT) ,
VAR I, J, INTEGER,
PROCEDURE P1,
VAR J, INTEGER,
    FUNCTION F (I2, INTEGER) , INTEGER,
        PROCEDURE P2 (VAR I1, INTEGER) ,
        BEGIN (*    P2    *)
            I1:=2*I1,
            WRITELN ('I1=', I1, 4)
        END,
    BEGIN (* F *)
        P2 (I2) , F:=I2*3,
        WRITELN ('F, J=', J, 4)
    END,
    BEGIN (*    P1    *)
        J:=100,
        J:=F (I) ,    WRITELN ('P1,J=', J, 4)
    END,
BEGIN (*    MAIN    *)
    I:=10,    J:=5,
    WRITELN ('I=', I, 4) ,
    P1,
    WRITELN ('J=', J, 4)
END.
(* I=10
   I1=20

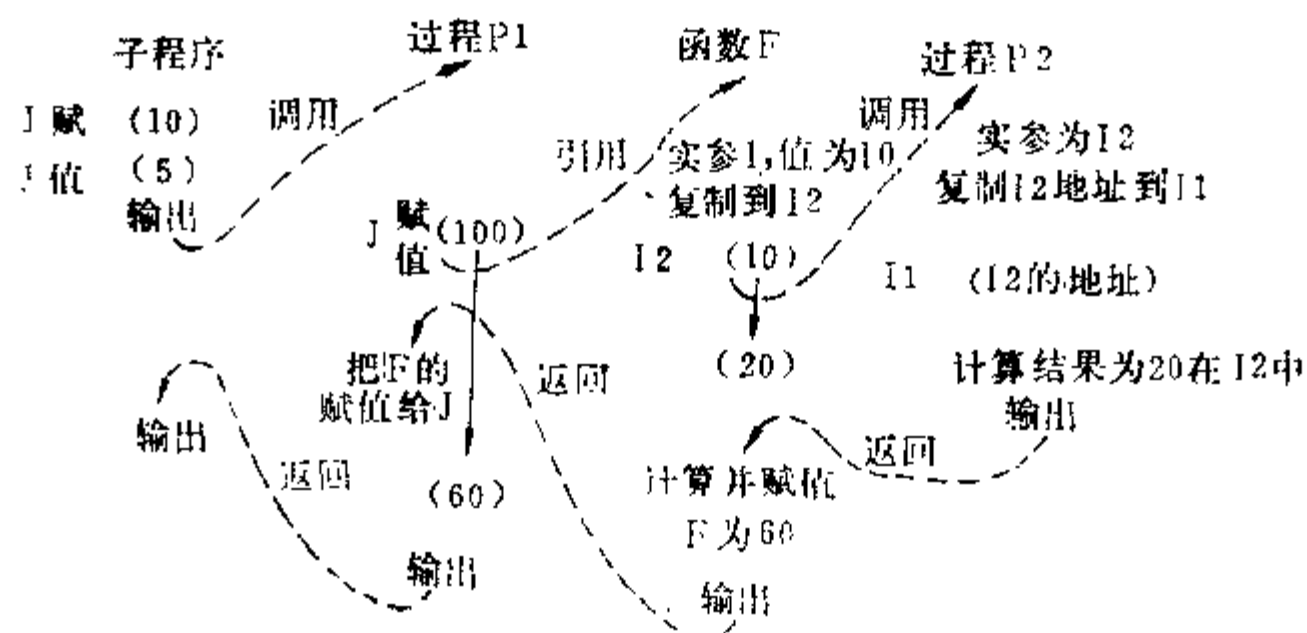
```



```

F:J=100
P1,J=60
J= 5    *)

```



第八个程序是应用并列过程,超前引用和间接递归调用的例子。这个程序中有两个并列过程P1和P2,它们之间存在彼此调用关系,因而形成P1调P2而P2又调P1的间接递归调用,对P2也存在间接递归调用问题。这里势必存在一个提前说明,如对P2的处理。在存在递归调用关系时,一定要处理好正确结束递归的问题,否则就出现死循环。这里的判断条件是在P2中判变量形式参数X是否小于400。

```

PROGRAM RECURS2 (INPUT, OUTPUT);
VAR A, B, N: INTEGER;
PROCEDURE P2 (VAR X: INTEGER); FORWARD;
PROCEDURE P1 (Y: INTEGER);
BEGIN
    Y := 2 * Y + 10;
    B := Y DIV 30;
    P2 (Y)
END;
PROCEDURE P2;
BEGIN
    IF X < 400
    THEN
        BEGIN
            N := N + 1;
            P1 (X)
        END
    END;
END;
BEGIN (* MAIN PROGRAM *)
    N := 0; READLN (A);
    P1 (A);

```

```

WRITELN ('N=', N:3, '    B=', B:4)
END.
(* 4
N=4    B=14
0
N=5    B=21
500
N=0    B=33    *)

```

当X小于400时，调过程P1，修改X（形式参数X必须为变量参数），直到X等于或大于400。这个程序中就潜伏着一个问题，试想当A的值为-10时，P1计算后仍得-10，就会出现死循环。当A小于-10时，如为-11，P1计算后Y为-12，越递归调用，X值越负，离400越远，就会形成Y值溢出错误。只有当A的值大于-10时，该程序才能有正确的运行结果。如输入为21和504时，输出分别为：

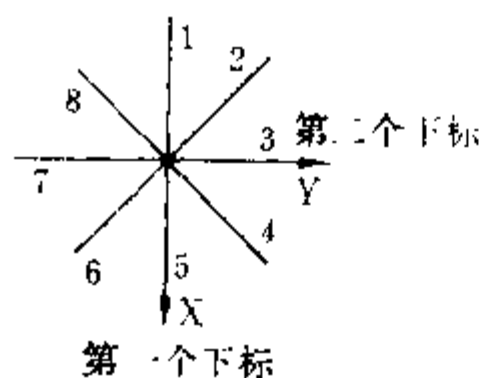
```

N=4    B=16
N=0    B=504

```

第九个程序是过程直接递归的例子，实现的是走迷宫的功能。用计算机找出迷宫路径的办法，就是八方试探。这里的迷宫是用10*10的一个矩阵表示的。每个座标点上的0表示路径能通，1表示不通。程序中试探路径的办法，是在与当前位置相邻的各座标点上找为0的座标点，找到后走到那里去，然后继续重复前一操作，在所有方向上都不能走，则退回一步，重新在还没试过的方位上继续试探。最后的结局，如果迷宫有通路，一定能走出迷宫，没通路，一定返回原出发点。这种问题特别适合用过程直接递归的办法解决，因为问题本身就是一个用递归方式定义出来的事物。

这个程序中有两个问题是必须解决好的。一是在迷宫中的走动情况怎样与10*10的矩阵对应起来，二是怎么检查迷宫边界条件。对第一个问题，把10*10的迷宫矩阵用一个10*10的二维的整型数组表示，每个座标点用MAZE[X,Y]表示，它可以取值为0或1。这样，在迷宫中走动的情况就可以用改变数组的下标值来表示了。与当前位置MAZE[X,Y]对应的八个方位上的相邻座标点分别为：



```

[X-1,    Y],      [X-1, Y+1]
[X    , Y+1],      [X+1, Y+1]
[X+1,    Y],      [X+1, Y-1]
[X    , Y-1],      [X-1, Y-1]

```

主程序中的数组MOVEX、MOVEY中给出的数值就是用来计算各相邻座标点对应的下标位置的。此外，凡是已走对的座标点，要标记出来，保证今后不重走已证明走不通的路，这是用把走过的座标点的取值由0改成4完成的。当走出迷宫后，还要把走通的路径上的各座标点的值再加1，即由4改为5。最后输出矩阵的内容如下图所示。

图上明确表示出了走过的（走通的与没走通的）路径，并用数字4和数字5区分清两

种情况。

1	1	1	1	1	1	1	1	1	1
5	5	5	5	4	4	4	4	1	1
1	1	5	1	1	1	1	1	4	1
1	1	1	5	1	1	1	4	1	1
1	1	1	1	5	1	1	4	4	1
1	1	1	1	5	1	1	4	1	1
1	1	1	4	5	5	1	1	4	1
1	1	1	5	1	1	1	1	1	1
0	1	1	1	5	1	1	0	1	0
1	0	1	5	1	1	1	1	1	1

走迷宫的结果

1	1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	1	1
1	1	0	1	1	1	1	1	0	1
1	1	1	0	1	1	1	0	1	1
1	1	1	1	0	1	1	0	0	1
1	1	1	1	0	1	1	0	1	1
1	1	1	0	0	0	1	1	0	1
1	1	1	0	1	1	1	1	1	1
0	1	1	1	0	1	1	0	1	0
1	0	1	0	1	1	1	1	1	1

迷宫初值

```

PROGRAM MAZE (INPUT, OUTPUT),
TYPE MOV=ARRAY [1..8] OF INTEGER,
CONST
  MOVEX=MOV (-1, -1, 0, 1, 1, 1, 0, -1),
  MOVEY=MOV (0, 1, 1, 1, 0, -1, -1, -1),
VAR
  I, J, INTEGER,
  SUCCESS, BOOLEAN,
  MAZE, ARRAY [1..10, 1..10] OF INTEGER,
  S, SET OF 1..10,
  FIN, FOUT, TEXT,
  CH, CHAR,
  PROCEDURE GO (X, Y, INTEGER),
  VAR
    L, X1, Y1, INTEGER,
  BEGIN
    MAZE [X, Y] := 4,
    ! writeln;      writeLn,

```

```

1  writeln ('---x, x=', x:4, y:4) ;
FOR L:=1 to 8 do
  IF NOT SUCCESS
    THEN
      BEGIN
        X1:=X+MOVEX [L] ,
        Y1:=Y+MOVEY [L] ,
        IF (X1 IN S) AND (Y1 IN S)
          THEN begin
            1 writeln ('    L=', L:2, x1:6, y1:4) ,
            IF MAZE [X1, Y1] = 0
              THEN GO (X1, Y1)
            end
          ELSE
            IF (X<>I) AND (Y<>J)
              THEN SUCCESS:=TRUE
            END
          ELSE BREAK;
        MAZE [X, Y] :=MAZE [X, Y] +ORD (SUCCESS)
      END;
END;
BEGIN (*MAIN PROGRAM*)
  ASSIGN (FIN, 'MAZE.MAZ') , RESET (FIN) ,
  ASSIGN (FOUT, 'RESULT.MAZ') , REWRITE (FOUT) ,
  FOR I:=1 TO 10 DO
    BEGIN
      FOR J:=1 TO 10 DO
        READ (FIN, MAZE [I, J]) ,
        READLN (FIN) ,
      END;

  S:= [1..10] , SUCCESS:=FALSE
  I:=2,    J:=1,    GO (I, J) ,
  Writeln, Writeln,
  FOR I:=1 TO 10 DO
    BEGIN
      FOR J:=1 TO 10 DO
        BEGIN
          WRITE (MAZE [I, J] :3) ,
          WRITE (FOUT, MAZE [I, J] :3)
        END;
    END;

```

```

        WRITELN; WRITELN (FOUT)
    END;
    CLOSE (FOUT);
    IF NOT SUCCESS THEN WRITELN ('-----NO PATH-----')
END.

```

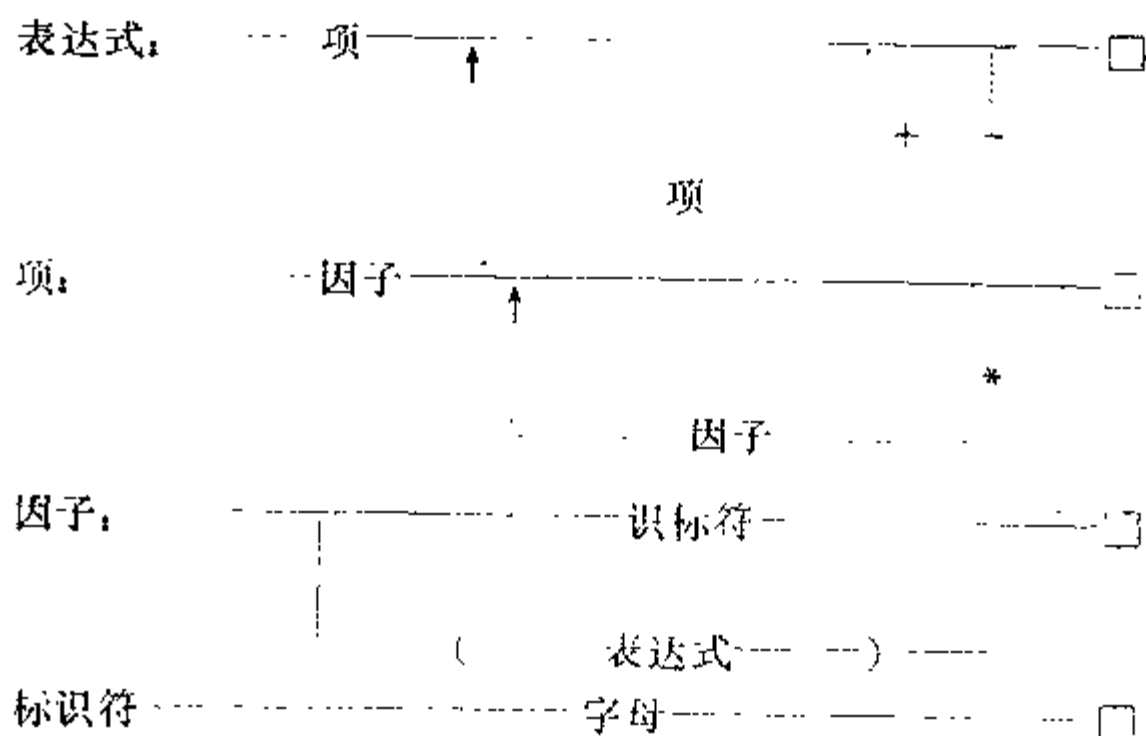
第二个问题是检查迷宫的边界条件。这里是通过给出的一个1..10 的集合变量进行检查的。只有当要试探的座标点下标值属于这个集合时，才进行试探，否则就是到了边界了。当此时的位置与出发点不同时，则说明走出了迷宫，否则表示迷宫没有通路（对于给定的出发点而言）。

这个程序中的完成走一步的过程GO 编写的很自然，几乎就是上面说的一段话的英文译稿，充分体现了过程递归调用的优点。过程中用了一个布尔变量SUCCESS，是为了配合过程中的 FOR 语句（每循环一次表示在一个方向上的一次试探）而使用的，一旦走出迷宫，就没有必要再在尚未试探过的方位上继续试探，而用BREAK语句提前结束当前的循环。

思考题

如果不用过程递归调用办法写这个程序，用户就必须记忆走过的路径，就要管理与使用一个堆栈，请试编出这个程序，并解释为什么用过程递归调用就见不到堆栈管理问题，系统本身是怎么解决的，请代替计算机走几步。

第十个程序是嵌套过程间形成的递归调用的例子。程序的功能是把从终端输入的一个由括号、*、+、和-组成运算符，用一个英文字母表示一个运算数而组成的代数表达式转换成它的逆波兰表达式。这是PASCAL 语言处理表达式的一种最简化的形式。这个程序中处理的表达式的语法图如下：



```

PROGRAM POSTFIX (INPUT, OUTPUT);
VAR
    CH: CHAR;
    PROCEDURE FIND;
    BEGIN

```

```

    READ (CH) ,
    WHILE CH=' ' DO READ(CH)
END,
PROCEDURE EXPRESSION,
VAR
    OP, CHAR,
    PROCEDURE TERM,
    PROCEDURE FACTOR,
    BEGIN
        IF CH=' ('
            THEN
                BEGIN
                    REPEAT
                        FIND; EXPRESSION
                    UNTIL CH=') ' ,
                    FIND
                END
            ELSE
                BEGIN
                    IF CH<>') '
                        THEN WRITE (CH) ,
                        FIND,
                    END,
                END,
        BEGIN (* TERM *)
            IF CH< >' * '
                THEN FACTOR,
            WHILE CH=' * ' DO
                BEGIN
                    FIND,
                    FACTOR,
                    WRITE (' * ')
                END
            END,
        BEGIN (* EXPRESSION *)
            IF (CH<>' + ') OR (CH<>' - ') THEN TERM,
            WHILE (CH=' + ') OR (CH=' - ') DO
                BEGIN
                    OP:=CH,    FIND,
                    TERM,    WRITE (OP)
                END
            END
        END
    END

```

```

            END
        END,
    BEGIN (* MAIN *)
        FIND,
        REPEAT
            EXPRESSION,
        UNTIL CH = ' '
    END.

```

```

    ( * (A + B) * (C - D) + E + F * G
      AB + CD - * E + FG * + * )

```

这里的递归调用是因为 EXPRESSION 调 TERM, TERM 调 FACTOR, 而 FACTOR 又可以调 EXPRESSION 而形成的。而 EXPRESSION, TERM 和 FACTOR 之间是嵌套结构。

当输入为 $(a+b) * (c-d)$

输出则为 $ab+cd-*$

当输入为 $b+c * (d+c * a * a) * b+a$

输出则为 $bc+dca * a * + * b * a +$

第十一个程序给出使用外部函数与外部过程的两个程序例子。第一个程序由两个文件组成。

```

MODULE PROCFUNC [PUBLIC] ,
PROCEDURE FACTORSUM (X: INTEGER; VAR Y: INTEGER) ,
VAR
    I: INTEGER,
BEGIN
    Y := 1,
    FOR I := 2 TO X DIV 2 DO
        IF X MOD I = 0 THEN X := Y + 1
    END,
FUNCTION SUMFACTOR (X: INTEGER) : INTEGER,
VAR
    I, J, INTEGER,
BEGIN
    J := 1,
    FOR I := 2 TO X DIV 2 DO
        IF X MOD I = 0 THEN J := J + 1,
    SUMFACTOR := J
END,
END.

PROGRAM EXTRNL (INPUT, OUTPUT) ,

```

```

VAR A, B, C, INTEGER,
    FUNCTION SUMFACTOR (X, INTEGER) , INTEGER, EXTERN,
    PROCEDURE FACTORSUM (X, INTEGER, VAR Y, INTEGER) ,
                                EXTERNAL,

```

```

BEGIN

```

```

    FOR A:=2 TO 1000 DO

```

```

        BEGIN

```

```

            FACTORSUM (A, B) ,

```

```

            FACTORSUM (B, C) ,

```

```

            IF (A=C) AND (A<>B)

```

```

                THEN WRITELN ('Output from proc, ', A:6, B: 6) ,

```

```

            B:=SUMFACTOR (A) ,

```

```

            IF (A=SUMFACTOR (B) ) AND (A<>B) THEN

```

```

                BEGIN

```

```

                    WRITELN ('Output from func, ', A:6, B:6) ,

```

```

                    WRITELN

```

```

                END

```

```

        END

```

```

    END.

```

```

    ( *

```

```

        Output from Proc, 220 284

```

```

        Output from func, 220 284

```

```

        Output from Proc, 284 220

```

```

        Output from func, 284 220

```

```

    * )

```

第二个程序由三个文件组成,

```

INTERFACE,

```

```

    UNIT FACTORSUM (PROC, FUNC) ,

```

```

    PROCEDURE PROC (X,INTEGER, VAR Y,INTEGER) ,

```

```

    FUNCTION FUNC (X,INTEGER) ,INTEGER,

```

```

END,

```

```

    ( * $INCLUDE, 'FACTORS. INF' * )

```

```

IMPLEMENTATION OF FACTORSUM,

```

```

    PROCEDURE PROC,

```

```

    VAR

```

```

        I,INTEGER,

```

```

    BEGIN

```

```

        Y:=1,

```

```

        FOR I:=2 TO X DIV 2 DO

```

```

            IF X MOD I=0 THEN Y:=Y+1

```



```

END,
FUNCTION FUNC,
VAR
    I, J:INTEGER,
BEGIN
    J:=1,
    FOR I:=2 TO X DIV 2 DO
        IF X MOD I=0 THEN J:=J+1
    FUNC:=J
END,
END.
(* $INCLUDE:'FACTORS.INF' *)
PROGRAM USEUNIT (INPUT, OUTPUT),
    USES FACTORSUM (PROC1, FUNC2),
VAR A, B, C: INTEGER,
BEGIN
    FOR A:=2 TO 1000 DO
        BEGIN
            PROC1 (A, B),
            PROC1 (B, C),
            IF (A=C) AND (A<>B)
                THEN WRITELN ('Output from Proc:', A:6, B:6),
            B:=FUNC2 (A),
            IF (A=FUNC2 (B)) AND (A<>B) THEN
                BEGIN
                    WRITELN ('Output from func:', A:6, B:6),
                    WRITELN
                END
        END
    END
END.
(*
    Output from proc:  220  284
    Output from func:  220  284
    Output from proc:  284  220
    Output from func:  284  220
*)

```

这两个程序都是求亲密数对，与本节开始给出的程序 2 和程序 3 有类似处，只是这里用到了外部过程和外部函数。第一个程序用到了在 MODULE PROCFUNC 中实现的过程和函数。第二个程序用的是在 UNIT FACTORSUM 实现的过程和函数。

第十二个程序的功能，是解决一个人带物过河的问题。问题是说，一个人带有一只

羊、一筐菜和一只狼要过河，但船上除了该人外，最多每次只能再带一样东西过河。而当人不在场的情况下，羊和菜在一起，羊要吃菜，狼和羊在一起，狼会吃羊。问怎样安排，人才可以安全地把三样东西都运过河去。给出这个例子的主要目的，是让读者理解，怎样才能把实际问题变成计算机能处理的内容。同时，看一看多数读者不大熟习的枚举和集合两种数据的用法。顺便看一看应用过程给程序设计带来的好处。

要解决这类问题，首先要确定数据结构。在类型定义部分，定义了由人、狼、羊和菜四个处理对象组成的枚举类型OBJECT，并定义了由该枚举类型构成的集合类型SETOBJ。

在变量说明部分，说明了S、S1、S2、D和T五个SETOBJ类型的集合变量，S用来表示在河的出发一岸还有的处理对象，过河前，人、狼、羊、菜都在该集合中，过河完成后，该集合为空。D表示在河的到达岸还有的处理对象，过河前，该集合为空；过河完成后，该集合中包括全部四个处理对象。S1和S2给出不允许出现的集合内容，用来表明羊和菜、狼和羊不能单独在一起的事实。T给出由四个对象组成的集合整体，用于判断过河完毕的条件。在变量说明部分，还说明了OBJECT类型的两个枚举型变量X和Y，用作FOR语句的循环控制变量。

在程序中用到的算法是，先从S集合中挑选一个对象，使剩下来的对象不会组成不允许出现的集合，并把选中的对象运过河去。到河对岸后，检查除了人之外的所有对象是否形成不允许出现的集合，是，还得选一个对象运回出发一岸并使留下来的对象都能安全。过河结束的条件是四个处理对象都在到达岸一侧，这可以用 $S=[]$ 或 $D=T$ 来判断。

运过河去或带回出发岸的每一个对象，均在程序运行过程中被显示在终端屏幕上，这是通过调用过程GO和BACK完成的。

选用枚举和集合类型数据来设计该程序，将给简化程序设计，提高程序的可读性带来明显的好处。这充分体现出PASCAL语言的优越性。

下面是该程序的源清单和运行结果。

```
PROGRAM SHIP (INPUT, OUTPUT),
  TYPE OBJECT = (MAN, WOLF, SHEEP, CABBAGE);
  SETOBJ = SET OF OBJECT;
  VAR S, S1, S2, D, T: SETOBJ;
      I, J, K: INTEGER;
      X, Y: OBJECT;
  PROCEDURE GO (X1: OBJECT);
  BEGIN
    I := I + 1;
    WRITE ('STEP', I, 1, ' ');
    CASE X1 OF
      WOLF : WRITE ('WOLF'),
      SHEEP : WRITE ('SHEEP'),
      CABBAGE: WRITE ('CABBAGE')
    END;
    WRITELN ('from source to destination 1')
```

END;

PROCEDURE BACK (Y1: OBJECT) ;

BEGIN

WRITE ('BACK', 1, 1, ' ') ,

CASE Y1 OF

WOLF: WRITE ('WOLF') ,

SHEEP: WRITE ('SHEEP') ,

CABBAGE: WRITE ('CABBAGE')

END;

WRITELN ('form destination to source ! ')

END;

BEGIN (* MAIN PROGRAM *)

S := [MAN, WOLF, SHEEP, CABBAGE] ,

S1 := [WOLF, SHEEP] ,

S2 := [SHEEP, CABBAGE] ,

D := [] , T := S, I := 0, Y := MAN;

REPEAT

FOR X := WOLF TO CABBAGE DO

IF (X IN S) AND (X <> Y) and (S - [man] - [X] < > S1)
and (S - [man] - [X] < > S2)

THEN [S := S - [MAN] - [X] ,

GO (X) ,

D := D + [MAN] + [X] , break] ,

IF (D <> T) and ((d - [man] = S [1] or (d - [man] = S2))

THEN

[FOR Y := WOLF TO CABBAGE DO

IF (Y IN D) AND (Y < > X) and (d - [man] - [y] < > S1)
and (d - [man] - [y] < > S2)

THEN [D := D - [MAN] - [y] ,

BACK (Y) ,

S := S + [MAN] + [Y] , break]

ELSE IF D < > T

THEN [Y := MAN, D := D - [Y] , S := S + [Y]]

UNTIL D = T

END.

STEP 1 SHEEP from source to destination!

STEP 2 WOLF from source to destination!

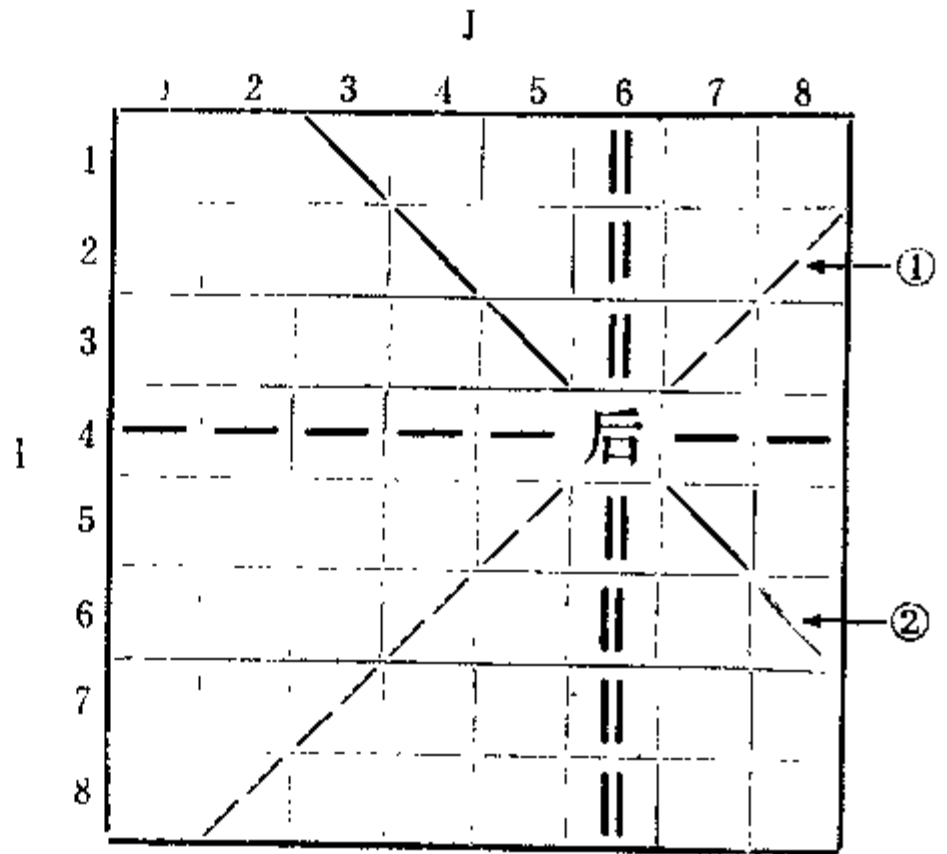
BACK 2 SHEEP form destination to source!

STEP 3 CABBAGE from source to destination!

STEP 4 SHEEP from source to destination

第十三个程序的功能是处理八皇后问题。就是在八行八列的棋盘上摆上八个皇后棋子，并使每一行、每一列、每个对角线方向上都只能有一个棋子，以保证不出现相互“吃子”的局面。

设计该程序的可用方案很多，这里给出的是一个设计得相当精巧的、采用过程直接递归调用方案的程序例子。算法中的关键，是把棋盘横向竖向八个格子按从1到8的顺序进行编号，按从上向下、从左向右的次序逐个位置找出可以放下一个棋子的格子，保证在同一横行、同一竖列和两个对角线方向上没有另一个棋子。如棋盘上所表示的。检查同一横向竖向的格子中是否已有其它棋子的方法是相对简单的，关键是要找出对角线方向上是否已有棋子，用的办法是考虑到在对角线方向上格子的座标编号的 $I+J$ 和 $I-J$ 具有相同的值，如棋盘所表示的，对角线①上的所有格子的 $I+J$ 的值均为10，对角线②上的所有格子的座标编号 $I-J$ 的值均为-2。从左上角到右下角共有15条对角线，其 $I+J$ 的值分别为2到16，从右上角到左下角共有15条对角线，其 $I-J$ 的值分别为-7到+7。程序中说明的B和C两个数组就被用来记忆在这些对角线上是否已有棋子。当I从1变到8时，数组X表示哪一列上已有棋子，数组A也用于检查哪一列上已有棋子。



下面是该程序的源清单和全部可能的12种结果的组合情况，其它结果都是这12种结果中的某种对称的变种形式

```
PROGRAM EIGHTQUEEN (OUTPUT);
VAR
  I: INTEGER;          Q: BOOLEAN;
  A: ARRAY [1..8] OF BOOLEAN;
  B: ARRAY [2..16] OF BOOLEAN;
  C: ARRAY [-7..7] OF BOOLEAN;
  X: ARRAY [1..8] OF INTEGER;
  PROCEDURE TRY (I: INTEGER; VAR Q: BOOLEAN);
    VAR J: INTEGER;
  BEGIN J:=0;
    REPEAT J:=J+1; Q:=FALSE;
      IF A[J] AND B[I+J] AND C[I-J] THEN
        [ X[I]:=J;
          A[J]:=FALSE; B[I+J]:=FALSE;
```

```

        C [ I - J ] := FALSE;
        IF I < 8 THEN
            [ TRY ( I + 1 , Q ) ,
              IF NOT Q THEN
                  [ A [ J ] := TRUE,
                    B [ I + J ] := TRUE,
                    C [ I - J ] := TRUE ]
              ]ELSE Q:=TRUE
            ]
        UNTIL Q OR ( J = 8 )
    END,
BEGIN {MAIN PROGRAM}
    FOR I := 1 TO 8 DO A [ I ] := TRUE;
    FOR I := 2 TO 15 DO B [ I ] := TRUE;
    FOR I := - 7 TO 7 DO C [ I ] := TRUE;
    TRY ( 1 , Q ) ,
    IF Q THEN
        FOR I := 1 TO 8 DO WRITE ( X [ I ] , 4 ) ;
    WRITELN
END.

```

1 —————>	1	5	8	6	3	7	2	4
2 —————>	1	6	8	3	7	4	2	5
3 —————>	1	7	4	6	8	2	5	3
4 —————>	1	7	5	8	2	4	6	3
5 —————>	2	4	6	8	3	1	7	5
6 —————>	2	5	7	1	3	8	6	4
7 —————>	2	5	7	4	1	8	6	3
8 —————>	2	6	1	7	4	8	3	5
9 —————>	2	6	8	3	1	4	7	5
10 —————>	2	7	3	6	8	5	1	4
11 —————>	2	7	5	8	1	4	6	3
12 —————>	2	8	6	1	3	5	7	4

最后两个程序是实现排序功能的例子。头一个实现的是多种方案的内排序，后一个程序实现的是对两个有序文件进行的外排序。

所谓排序，是指把按数据值的随意次序组织起来一批数据，变成按数据值大小依次排列的顺序关系。内排序是说把要排序的全部数据都首先调入主存，在主存内直接完成排序过程。内排序的运行速度快，但被排序的数据量受可用内存空间的约束。外排序是对保存在外存设备上的一批数据进行排序。而我们这里给出的例子，只是一种特定的对两个已分别排好序的磁盘文件的内容进行按序归并的操作。排序是计算机处理大量数据的过程中用得很多的操作，许多人曾进行了大量的研究，提出众多实现算法。排序本身不是本书要讨

论的内容，我们这里只给出程序清单和某种运行结果，不进行更多的解释。

内排序程序中给出了7种不同的排序算法，并按10种不同数据量给出每种算法排序所用的时间。结果给出在程序下边。

```
program sort (input, output),
! const n=1000;
var ar: array [0..1000] of integer;
x, l: integer;          tit, n: integer;
k: integer;             lst1, lst2: string (10);
w1, w2: word; r: real;
f: text;
procedure time (var s: string); extern;
function tic: word; extern;
function tim (var lst1, lst2: string): integer; extern;
procedure timecal;
begin
    r:=tim (lst1, lst2) *word (w2--w1) /100;
    if r<0 then [ r:=-r; writeln (lst1, lst2, w1, w2) ];
    write (f, r: 8: 3)
end;
function random: integer;
begin
    k:= (13077*k+6925) mod 32768;
    random:=ord (loword (k))
end;
procedure ini;
begin k:=0;
    for l:=1 to n do ar [l] :=random
end;
procedure print;
    var i: integer;
begin
    for i:=1 to n do
        [ write (ar [i], ' ');
          if i mod 10=0 then writeln
            ]; writeln
end;
procedure straightinsertion;
    var i, j: integer;
begin
    for i:=2 to n do
```

```

    [ x := ar [ i ] ; ar [ 0 ] := X ; j := i - 1 ;
      while x < ar [ j ] do
        [ ar [ j + 1 ] := ar [ j ] ; j := j - 1 ] ;
        ar [ j + 1 ] := x ;      ! print
      ]
    end ;
  procedure straightselection ;
    var i , j , k ; integer ;
  begin
    for i := 1 to n - 1 do
      [ k := i ; x := ar [ i ] ;
        for j := i + 1 to n do
          if ar [ j ] < x then
            [ k := j ; x := ar [ j ] ] ;
          ar [ k ] := ar [ i ] ; ar [ i ] := x ;      ! print
        ]
      end ;
    procedure bubblesort ;
      var i , j ; integer ;
    begin
      for i := 2 to n do
        [ for j := n downto i do
          if ar [ j - 1 ] > ar [ j ] then
            [ x := ar [ j - 1 ] ; ar [ j - 1 ] := ar [ j ] ; ar [ j ] := x ] ;
            ! print
          ]
        end ;
      procedure shakersort ;
        var i , j , k , r ; integer ;
      begin
        l := 2 ; r := n ; k := n ;
        repeat
          for j := r downto l do
            if ar [ j - 1 ] > ar [ j ] then
              [ x := ar [ j - 1 ] ; ar [ j - 1 ] := ar [ j ] ;
                ar [ j ] := x ; k := j ] ;
            l := k + 1 ;      ! print
          for j := 1 to r do
            if ar [ j - 1 ] > ar [ j ] then
              [ x := ar [ j - 1 ] ; ar [ j - 1 ] := ar [ j ] ;

```

```

        ar[ j ] := x; k := j ];
    r := k - 1;      ! print
until l > r
end;
procedure shellsort;
    const t = 4;
    var i, j, k, s: integer; m: 1..t;
        h: array[ 1..t ] of integer;
begin
    h[ 1 ] := 9; h[ 2 ] := 5; h[ 3 ] := 3; h[ 4 ] := 1;
    for m := 1 to t do
        [ k := h[ m ]; s := -k;
        for i := k + 1 to n do
            [ x := ar[ i ]; i := i - k;
            if s = 0 then s := -k;
            s := s + 1; ar[ s ] := x;
            while x < ar[ j ] do
                [ ar[ j+k ] := ar[ j ]; j := j + k ];
            ar[ j+k ] := x; !print
            ]
        ]
    end;
procedure heapsort;
    var i, r: integer;
    procedure sift;
        label l3;
        var i, j: integer;
    begin
        i := 1; j := 2 * i; x := ar[ i ];
        while j <= r do
            [ if j < r then
                if ar[ j ] < ar[ j+1 ] then j := j + 1;
            if x >= ar[ j ] then goto l3;
            ar[ i ] := ar[ j ]; i := j; j := 2 * i;
            ];
        l3: ar[ i ] := x; !print
    end;
begin
    i := n div 2 + 1; r := n;
    while i > 1 do

```



```

    [ i := i - 1; sift ];
while r > 1 do
    [ x := ar[ i ];    ar[ i ] := ar[ r ];    ar[ r ] := x;
      r := r - 1; sift ]
end;
procedure quicksort;
  procedure sort ( l, r, integer );
    var i, j, integer; w, integer;
  begin
    i := l; j := r;
    x := ar[ ( l + r ) div 2 ];
    repeat
      while ar[ i ] < x do i := i + 1;
      while x < ar[ j ] do j := j - 1;
      if i <= j then
        [ w := ar[ i ]; ar[ i ] := ar[ j ]; ar[ j ] := w;
          i := i + 1; j := j - 1
        ]
      until i > j;
      if l < j then sort ( l, j );
      if i < r then sort ( i, r )
    end;
  begin
    sort ( 1, n )
  end;
begin
  assign ( f, 'sortime.dat' ); rewrite ( f );
  for ttt := 1 to 10 do
    [ write ( f, ttt, 4 );
      n := ttt * 100;
      ini; time ( lst1 ); w1 := tics;
      straightinsertion;
      w2 := tics; time ( lst2 ); timecal;
      ! writeln ( lst1, lst2, w1, w2 );
      ini; time ( lst1 ); w1 := tics;
      straightselection;
      w2 := tics; time ( lst2 ); timecal;
      ! writeln ( lst1, lst2, w1, w2 );
      ini; time ( lst1 );
      bubble sort;

```

```

        w 2 := tics;    time (lst 2) ;    timecal;
! writeln (lst 1, lst 2, w 1, w 2) ;
    ini; time (lst 1) ;
        shakersort;
        w 2 := tics;    time (lst 2) ;    timecal;
! writeln (lst 1, lst 2, w 1, w 2) ;
    ini; time (lst 1) ;
        shellsort;
        w 2 := tics;    time (lst 2) ;    timecal;
! writeln (lst 1, lst 2, w 1, w 2) ;
    if ttt=1 then print;
    ini; time (lst 1) ;
        heapsort;
        w 2 := tics;    time (lst 2) ;    timecal;
! writeln (lst 1, lst 2, w 1, w 2) ;
    if ttt=7 then print;
    ini; time (lst 1) ; w 1 := tics;
        quicksort;
        w 2 := tics;    time (lst 2) ;    timecal;
! writeln (lst 1, lst 2, w 1, w 2) ;
    writeln (f)
],
close (f)
end.

```

1	0.110	0.220	0.450	0.040	0.150	0.010	0.060
2	0.380	0.880	1.090	1.080	0.410	0.320	0.110
3	0.880	1.980	2.620	1.810	0.310	0.310	0.170
4	1.480	3.460	4.020	2.810	0.520	0.120	0.270
5	2.260	5.430	7.460	6.120	1.050	0.820	0.320
6	3.300	7.800	9.910	7.090	0.300	0.230	0.330
7	4.550	10.540	13.280	10.370	0.910	0.050	0.440
8	5.710	13.780	17.520	14.530	2.400	1.660	0.500
9	7.470	17.420	21.990	17.120	1.420	0.850	0.600
10	9.120	21.480	27.210	21.400	2.040	0.630	0.660

程序中，七种排序算法是用 7 个过程实现的，排序所用的 10 种不同的数据量为 100 到 1000。被排序数据保存在 AR 数组中，并在该数组内完成排序。

为了比较不同排序算法的效率，我们让它们对同一批数据进行排序。每一数据是用一个随机数产生过程 RANDOM 得到的。INIT 过程按所要求的数据量多次调用 RANDOM 过程得到被排序的数据。

过程 TIME 和 TICS 是系统提供的测试系统时间和百分之一秒计算值的过程和函数。

TIME函数把两个用字符串表示的时间之差变为以秒为单位的整数值。过程TIME CAL用于求出一次排序过程所用的时间。

源程序中的感叹号表示它之后的本行内容为注释。感叹号是 MS PASCAL 中的单行注释符。

实现两个有序文件内容按序归并的程序的源清单如下:

```
PROGRAM MERGE (INPUT,OUTPUT);
VAR
  F1,F2,F3:FILE OF INTEGER;
BEGIN
  ASSIGN (F1,'A.DAT');   RESET (F1);
  ASSIGN (F2,'B.DAT');   RESET (F2);
  ASSIGN (F3,'AB.DAT');  REWRITE (F3);
  WHILE NOT EOF (F1) AND NOT EOF (F2) DO
    [ IF F1^ <=F2^ THEN [ F3^:=F1^;GET (F1) ]
      ELSE [ F3^:=F2^;GET (F2) ] ;
      PUT (F3) ];
  WHILE NOT EOF (F1) DO
    [ F3^:=F1^;PUT (F3); GET (F1) ];
  WHILE NOT EOF (F2) DO
    [ F3^:=F2^;PUT (F3); GET (F2) ];
  CLOSE (F1);CLOSE (F2);CLOSE (F3)
END.
```

这个程序容易读懂。当两个文件均未结束时,按条件 $F1^ \leq F2^$ 决定把哪个文件的记录内容写入文件F3,并接着读来它的下一条记录。

当有一个文件已经结束,则表明该文件的最后一条记录肯定已写入F3文件。对尚未结束的文件,首先要把已读出但未写入F3文件的记录内容写入F3,接着取下一条记录,完成写操作,直到文件结束。

习 题

1. 程序设计中自顶向下的方法指的是什么意思? PASCAL 语言中的过程和函数在这个方面起什么作用? 并列和嵌套过程体现的各是什么功能?

2. 解释过程、函数嵌套和递归调用的概念,说明嵌套过程在彼此调用和使用变量两个方面的有关规则。

3. 过程和函数都可以带有参数,这对程序设计有什么好处? 这些参数被分为哪四类? 数值参数和变量参数在用法上有什么区别?

4. 什么是外部过程和外部函数? 在说明和进行编译的方法上,使用外部过程与内部过程有什么区别?

5. 说明在本章第六节中给出的第一个程序中使用过程FACTOR的好处,再把它改成带有一个逻辑型参数的过程,用来表明能否求出变量X的因子积的表示形式。

6. 解释在求亲密数对的两个程序AMICABLE1和AMICABLE2中,FACTOR过

程为什么要用两个参数，而且K要被说明为变量参数？而FACTOR函数却可以只用一个形式参数呢？为什么两个程序的主程序的外貌相差那么大？把AMICABLE1程序改写成不用GOTO语句的形式，并保持输出结果与原来完全相同。

7. 在给出的实现直接递归调用的程序RECURSION中，为什么要检查N大于零和小于8这样的条件？当把函数F的结果类型变为实数类型时，对这个判别条件会有什么影响吗？

8. 把程序MAZE改写成不用过程递归方法实现的走迷宫的程序。对比这两个程序说明使用过程递归调用的好处。

9. MS PASCAL语言给出哪些用于扩展语言功能的过程和函数？

10. UNIT和MODULE向用户提供过程、函数的方式有何不同？说明它们在支持结构程序设计方面的作用。

11. 按你的理解，总结MS PASCAL语言为结构程序设计所提供的支持（请不要忘记在数据结构方面的有关内容）。

第六章 屏幕上格式数据的输入和输出支持

在使用和实现数据库管理系统的过程中,在设计其它大型综合软件的过程中,总会遇到在计算机的终端屏幕上显示入/出的格式数据的问题。这里说的屏幕上的格式数据,主要有如下六项要求:

- 从屏幕的什么行列位置开始,显示多少个输入或输出的字符;

- 输入或输出的字符按什么属性 (ATTRIBUTE) 进行显示 (如正常属性、高亮度或闪烁以及所用颜色等等);

- 允许每一个输入的数据项接收哪些种类的字符;

- 在一屏数据输入和输出的过程中,能使用哪些功能键;

- 在处理一屏数据输入的过程中,能用类似于全屏编辑的方式输入数据,也就是说,可以通过移动光标 (CURSOR) 的办法,随时修改或输入一个字符到光标位置,并相应修改有关变量的内容;

要求输入/输出操作的一屏数据的逻辑屏幕的范围可以比终端实际屏幕的范围 (一般为25行×80字符) 大,譬如66行×132字符,为一张大的打印纸的大小。此时要能方便地上下左右移动屏幕上显示的内容。

在这六项要求中,为满足第五项需要解决某些技术难点。而第六项要求,则需要用户按给出软件所提供的基本功能,结合自己的实际需要自行处理;需要时,我们可以实现一个使用更方便、能更灵活地支持上下左右移动屏幕、甚至支持“多窗口”屏幕显示的软件工具。

本章内容介绍实现屏幕上格式数据入/出管理的基本知识和原理性技术。如果把这些实现了的软件功能做到 MS PASCAL 语言的一个模块 (MODULE) 或单元 (UNIT) 中,则任何一个 PASCAL 的程序都能够灵活方便地使用它们,从而达到简化管理屏幕上格式数据输入与输出的目标。

6.1 终端硬件特性及其使用方法

IBM PC 机、GW 0320 机或 WANG PC 机的终端,都提供了使用终端的某些硬件支持。不少非计算机专业的用户却很少了解这些内容。还有更多的用户,总认为无法在 PASCAL 的程序中直接使用它们。这是一种误会。了解终端提供的硬件支持,是一件很容易的事,并不要求过多的其它知识;而在 PASCAL 的程序中直接使用这些硬件支持,更是一目了然的事情。说明这些问题,为的是先打消某些初学者的顾虑,增加一点学好用好本章知识的兴趣。

终端提供的硬件支持,与本章内容直接有关的都是用一个字节的控制码和 ANSI ESC 序列码组成的控制串实现的。这些控制码和控制串会由 DOS 操作系统中的 BIOS 软件识别,它们仅用来完成对终端屏幕的显示控制。这些控制包括:

- 响铃 使终端里的喇叭发出一次响声;

- 清屏 清除整个屏幕上或部分屏幕上显示的内容;

- 回车、进行 (进行还可能带有“退”行);

- 移光标 光标上下左右移动;
- 设置终端本身的特性, 如一屏上每行中字符的个数, 几个指示灯的亮灭等等;
- 设置终端在某个位置上的显示属性, 如高亮度、下横线、上榜标、下脚标、闪烁、灰度以及在彩色终端上选择颜色等;
- 字符的插入与删除;
- 行的插入与删除;
- 光标位置的保留和恢复, 光标的停还是动, 光标显示还是不显示等等;
- 屏幕上行的滚动。

终端的一个字节的控制码共有七个。我们用十六进制和十进制形式给出它们的编码值和它们所完成的控制功能, 列表如下:

编 码 · 值		完 成 的 控 制 功 能
1 6 # 7	7	响铃一次
1 6 # 8	8	光标左移一次, 光标位置上显示的字符不变
1 6 # 9	9	光标右移一次, 光标位置上显示的字符不变
1 6 # 0 A	10	光标进到下一行, 需要时伴随有滚行
1 6 # 0 C	12	清除屏幕上显示的全部内容, 光标移到屏幕的最左上角位置。
1 6 # 0 D	13	光标移到光标所在行的行首
1 6 # 1 B	27	用来组成 ANSI ESC 序列的控制串

要在 PASCAL 的程序中使用这些控制字符, 只需用一个 WRITE 语句, 把这个控制字符的编码值作为一个字符输出就行了。例如:

WRITE (CHR (07)); WRITE (CHR (12)) 分别实现响一次铃和清屏操作。这里使用 CHR 函数是绝对必要的, 因为控制字符不是可打印的字符, 无法用直接输出字符的方式输出。也不能用直接输出控制字符的编码值的方式输出, 否则会把它当成一个整型量写到屏幕上, 达不到控制目的。只能用 CHR 函数完成控制字符的发送操作。

下面介绍由 ESC 开头组成的控制串。ESC 是终端键盘上的一个键, 但它不对应任何一个可打印的字符, 它也是一个控制字符。它的用法有些特殊, 它只被用来组成对终端(或打印机)的控制串, 计算机的操作系统能识别它, 并按约定规则处理跟在它后面的一个或多个字符, 把它们处理成一个对终端的控制命令。因此当用户先按一次 ESC 键再按另外的键时, 可能构成一个对终端的控制命令, 而不是把跟在 ESC 键之后的一个或几个字符与正常情况下输入的字符一样处理。因此, 有些人习惯上称 ESC 键为“逸脱键”。由 ESC 开头构成的控制串是控制终端特性的重要手段。可按它们完成的功能分类综述如下:

1. 控制终端显示特性 ESC [p, m

这里的 ESC 代表 ESC 键 (通常用 CHR (27) 给出), [和 m 是跟在 ESC 之后的字符 ' [' 和 ' m '。小写的 p, 代表的是由 1 或两个数字符组成的不带符号的正整数。这个数值取不同的值时, 实现不同的显示特性或不同的显示颜色, 具体规定如下:

p, 的值	显 示 特 性
0	正常属性
1	高亮度, 即比正常亮度要亮一些
2	下脚标, 即把显示的字符作为下脚标显示
3	上脚标, 即把显示的字符作为上脚标显示
4	在显示的字符下边, 加一条下横线
5	闪烁, 使显示的内容一亮一暗地闪烁
6	在显示的字符上边, 加一条上横线
7	反底色, 屏幕底色和显示的字符对调颜色
8	隐藏, 即不把正常情况下应显示的字符显示到屏幕上
30-31	控制不同的颜色 (只适用于彩色终端)

要在 PASCAL 的程序中使用这类控制串, 可以用一个 WRITE 语句向终端输出控制串的内容。例如:

WRITELN (CHR (27), ' [1 m ', ' 高亮度显示 ', CHR (27), ' [0 m '), 就把要输出的串内容“高亮度显示”几个汉字, 以较正常显示亮度更亮的方式显示在终端屏幕上。此处的 WRITELN 语句与输出其它内容的 WRITELN 语句完全相同, 只是在此处正常输出串的前后都加入了显示的属性控制。前面的 CHR (27) 是把 ESC 键的编码值当成一个字符的编码变换成字符类型的量, 接在它后面的 ' [1 m ' 串表明要实现高亮度显示。最后的也是一个显示特性控制, 即恢复终端为正常显示特性, 以保证后面的输出用正常的特性显示。其它的显示特性的控制办法与上述例子完全相同。

这里还要说明两点。第一, 在需要同时使用几个显示特性时, 可以在同一个控制串中把 p 的数值用分号隔开。例如 WRITE (CHR (27), ' [1; 5 m ', ...) 就同时使用了高亮度和闪烁两种显示特性。这种用法在使用彩色终端时更有用。例如, 在给出显示特性的同时, 指明选用的颜色。例如, WRITE (CHR (27), ' [4; 45 m ', ...)。CHR (27), ' [4; 45 m ' 与 CHR (27), ' [1 m ', CHR (27), ' [45 m ' 的效果是完全一样的。要说明的第二点是, 这些控制串通常是用字符串形式输出到终端的。串内的英文字母的大小写一定要按命令串的规定使用, 不能随意改动。例如, 控制显示特性的串中用的是小写字母 m, 当写成大写字母 M 时就错了。此外, 在上面的例子中的 p, 值都给出常量值, 其实也可以使用变量值, 只是在输出之前, 先用 DECODE 函数把有关变量的值, 变换

成长度为 2 的 LSTRING 类型的变量，再在 WRITE 语句中使用这个 LSTRING 类型的变量就行了。例如，假定 L21 和 L22 都是 LSTRING (2) 类型的变量，用 WRITE (CHR (27), '[', L21, ',', L22, 'm') 语句也能实现控制终端显示特性的要求。

后面给出的 ESC 序列的用法，与上面讲到的很类似。我们将只给出每个序列的组成格式和它们的控制功能，不再一一说明它们的用法。

2. 控制光标位置

ESC [p_n 控制光标移动

这里的 p_n 代表一个大写的英文字母，它只取 A、B、C 或 D 四个大写英文字母中的一个。具体规定如下：

p _n 的值	控 制 功 能
A	光标上移一次
B	光标下移一次
C	光标右移一次
D	光标左移一次

(1) ESC [P_r; P_c^H 控制光标定位

这里的 p_r 和 p_c 分别为由两个数字字符组成的正整数，分别表示屏幕上某一位置的行号和列号。该串功能用大写字母 H 或小写字母 f 实现，二者无任何差异。p_r 的值应在 1 到 25 之间，p_c 的值应在 1 到 80 之间（或 1…40 之间）。这个串的功能是把光标定位到由 p_r 和 p_c 指定的屏幕位置上去。例如 WRITE (CHR (27), '[12; 60H') 语句就实现把光标定位到屏幕的第 12 行第 60 列的位置。请注意，此处一定不能用 WRITE LN 语句实现，因为这个语句完成定位光标的操作之后，还要执行回车进行的功能，从而使光标又从第 12 行的 60 列的位置移到了第 13 行第一列的位置。屏幕上行的编号是从上向下编为 1 到 25，列的编号是从左向右编为 1 到 80。

(2) ESC [s_s 保存、恢复光标位置

这个串用于保存与恢复光标位置。小写字母 s 表明保存光标当前位置。小写字母 u 表明把光标恢复到原来保存下来的初始位置上去。在对屏幕上信息正常操作的过程中，暂时要跳到屏幕另一个位置上输出一句提示信息后，再返回去继续原来的操作过程，这是非常有用的两个命令。

(3) 清屏控制

ESC [p,J 清除整个屏幕或部分屏幕上显示的内容

这里的 p_r 可以是 2、1 或 0 三个数字字符中的一个字符。为 2 时，清全屏，为 1 时，清光标之前的前半屏，为 0 时，清光标之后的后半屏。ESC [2J 与 16# 0C 控制码的作用相同。这里用的是大写字母 J。

(4) 清行控制

ESC [p,K 清除光标所在行整行或半行的显示内容

这里的 p_r 与清屏控制串中的 p_r 意思相同，也只能取 2、1 或 0 三个数字字符中的一个。

为 2 时，清全行；为 1 时，清光标之前的半行；为 0 时，清光标之后的半行（包括光标所在位置的字符）。该串中用的是大写字母 K。

(5) 行插入或行删除控制

ESC [p_n; p_rL

ESC [p_n; p_rM

插行的控制命令串

删除行的控制命令串

大写的 L 指明要插入一些行，大写的 M 为删除一些行。这里的行插入或行删除操作与上面的清行控制是不同的。清行只是把光标所在行上显示的内容清除，使本行变成一个空行，或变成半空行（只半行有内容）的局面，它并不影响在该行上面的或下面的各行的位置和内容。行删除和行插入则不一样，它还会影响到上面的各行或下面的各行执行向上或向下的滚动操作。

这里的 p_n 代表所要插入或删除的行数。p_r 用于表示是上面的一些行还是下面的一些行跟着滚动。若没有用 p_n 明确给出具体行数值，则默认其值为 1，执行删除光标所在行，或在光标所在行位置插入一个新的空行。p_r 的值，可以取零或非零两类值。若省略 p_r 的值，则默认它的值为零。关于 p_r 的值和它实现的控制的具体规定是：

命 令 串	p _r 的 值	
	0 值（为默认值）	非 0 值
行 插 入	下滚下半屏各行	上滚上半屏各行
行 删 除	上滚下半屏各行	下滚上半屏各行

例如，WRITE (CHR (27), '[L') 语句实现的是在光标所在行位置插入一个空行。原来光标所在行的内容和后续各行内容都下移一行，原来屏幕上最下边一行的内容不能显示了。这是取 p_n 的默认值为 1，p_r 的默认值为 0 的例子。WRITE (CHR (27), '[3; 1M') 完成的是删除光标所在行和接在它下面的两行共计三行的内容，并使光标所在行之上的各行内容往下移动三行，此时屏幕最上边的三行变为空行。

(6) 控制删除字符

ESC [p_nP

这里的 p_n 是要删除的字符个数，被删除的字符是从光标所在位置的字符向左数起，共 p_n 个，这之后的字符将向左移动 p_n 个位置，与前边的字符接上，光标位置保持不动。该串功能用大写字母 P 来实现。

(7) 控制屏幕滚动

ESC/p₁; p₂; p₃; p₄S

该串用到四个参数，并用到字符“/”和大写字母 S。四个参数所代表的含义是：

参 数	作 用	默 认 值
p ₁	滚动的起始行号	无默认值，必须明确给出
p ₂	滚动的截止行号	无默认值，必须明确给出
p ₃	滚动的行数	1
p ₄	滚动的方向 0：下滚， 1：上滚	1

这个控制串，主要用于选定屏幕上一块确定区域完成显示信息的目的。在此区域内工

作时，不会影响未选区域的显示内容。在实现屏幕上多窗口技术时，该控制串特别有用。

(8) 设置或清除终端的工作方式，控制LED指示灯

ESC [P,^h_l]

小写字母 h 表示设置

小写字母 l 表示清除

该控制串的功能比较多，是用 p_l 的值来表明控制功能的，具体规定如下：

p _l 的值	控 制 功 能	
	h	l
4	进入插入方式	退出插入方式
5	不显示光标	显示光标
6	支持软滚动	不支持软滚动
7	一行80个字符	一行40个字符
8	光标动	光标停止不动
9	回车带进行	回车不带进行
10	点亮指示灯 0	熄灭指示灯 0
11	点亮指示灯 1	熄灭指示灯 1
12	点亮指示灯 2	熄灭指示灯 2
13	点亮指示灯 3	熄灭指示灯 3
14	点亮指示灯 4	熄灭指示灯 4
15	点亮指示灯 5	熄灭指示灯 5

指示灯 0, 1, 2, 3, 4, 5, 是终端键盘上的六个指示灯。可以用这个命令串使每个灯点燃或熄灭。

p_l 取值为 6 和 7 时实现的控制只适用于低分辨率的终端。

对该控制串中涉及到的几个概念，要简单做一些说明。

终端的插入方式和非插入方式，是指当从键盘输入一个字符时，处在屏幕光标位置上及本行中它后边的字符是向右移一列，以便把新输入的字符显示在刚“腾”出来的位置上，还是用刚输入的字符，简单地取代光标位置上原来显示的字符。这两种显示方式经常使用，可以用该控制串实现两种方式之间的切换，并相应的点燃或熄灭指明插入方式操作的指示灯。

关于光标控制的功能，有显示还是不显示，让光标随着显示的字符移动还是静止不动两大类控制，这些也都属于常用方式。在我们正常使用终端时，采用的是显示光标和让光标移动的方式。

关于回车带不带进行的问题，一般应选用回车带进行的方式，保证按了一次 RETURN 键之后，光标进到下一行的行首，而不是本行行首。

这些控制串的应用场合和使用效果，将在后面给出的程序例子中进一步解释。

(9) 恢复终端状态到初始设定的状态

ESC [z

这里使用的是小写字母 z。该控制串的功能，是使终端状态恢复到初始设定的标准状态。当用户在自己的程序中改变过终端工作状态时，在程序结束之前，应该用该控制串恢复终端的状态，以避免用户程序结束后，终端无法正确运行的错误。

我们可以把前面给出的控制串，汇总到一张表格中，以便快速查询有关内容。

命令串格式	主要控制功能	有关参数的规定
ESC [p _i m	控制终端的显示特性	p _i 取值'0'到'90'
ESC [p _a	控制光标移动	p _a 取值'A'到'D'
ESC [p _r ; p _c ^H	定位光标	p _r 取值'1'到'25'
ESC [^s _u	保存光标当前位置 恢复光标原始位置	p _c 取值'1'到'80'
ESC [p _s J	清屏	无参数
ESC [p _s K	清行	p _s 取值'0','1','2'
ESC [p _n ; p _s L	行插入	默认值为'0'
ESC [p _n ; p _s M	行删除	p _s 可取零值或非零值
ESC [p _n P	字符删除	p _s 默认值为'0'
ESC/p _s ; p _e ; p _c p _d S	控制屏幕上滚动特性	p _n 的默认值为'1'
ESC [p _s ^h	设置或清终端工作方式，燃或灭指示灯	p _e 的默认值为 1 p _d 的默认值为 0
ESC [z	恢复终端正常状态	p _s 取值'4'到'15'
		无参数

对彩色终端，可以选用不同的底色(background Colour)和字符显示色(foreground-colour)。由于彩色终端有不同的分辨率(resolution)，因此能提供的颜色种类和亮度级别也不同，用户在使用不同的彩色终端时，应查询有关用户或技术手册。选择颜色是用 ESC 命令串完成的，与选择单色终端的显示属性的 ESC 命令串类似，只是所用的数字字符不同。以八种底色和八种显示色为例，通常把 ESC [之后的 30—37 用于选底色，40—47 用于选显示色，在串的不同数字字符组之间用分号隔开，最后面用小写字母 m 结束。例如：

chr (27) , ' [30; 45m' 和 chr (27) , ' [42; 37m' 都是正确的命令串。在使用中，要注意底色和显示色不应相同或太接近，否则是显示的字符看不见或看不清。ESC , ' [0 m' 还起恢复正常显示属性的作用。

下面是一个表明如何选择不同底色与显示色的小程序。该程序的功能，是按用户输入的文件名打开该文件（字符文件，即 TEXT 类型），然后用不同底色和显示色把文件中的每一行显示在终端屏幕上。

```

program abc (input, output) ,
var buff: lstring (81) ,
    esc: char;      fname: lstring (10) ,

```

```

    fcolor, bcolor, count: integer;
    f: text;
begin
    esc:=chr(27);
    write('Giving the file name : ');    readln (fname);
    assign(f, fname); reset (f);
    write(esc, '[2I');
    count:=0;    fcolor:=0;    bcolor:=3;
    while not eof (f) do
        [readln (f, buff);
        fcolor:=fcolor+30;
        bcolor:=bcolor+40;
        writeln(esc, '[',bcolor:2, '; ', fcolor:2, 'm', buff);
        fcolor:=(fcolor+1) mod 8;
        count:=(count+1) mod 7;
        if count=0 then bcolor:=(bcolor+1) mod 8;
        if fcolor=bcolor then fcolor:=(fcolor+1) mod 8;
        ];
    close (f); write(esc, '[ 0 m')
end.

```

在这个程序中，希望底色和显示色都是按 8 循环变化，并且，为了使各种不同的底色和显示色有机会搭配出现，每显示 7 行使底色值多计数一次。为了不使底色和显示色相同，要进行必要的检查，发现相同时，使显示色的值增 1。

下面给出在 PASCAL 程序中使用某些终端控制串的例子。这几个小程序，既用来表明使用命令串的格式和取得的效果，同时也具有一定实际应用的参考价值。换句话说，对给出的小程序进行某些补充和完善，都可以使其成为有真正使用价值的程序。

第一个程序，主要用到了清屏，光标定位，控制终端显示特性和响铃等功能。程序清单如下：

```

program boel (input, output);
label 1;
var i, j: integer;
    esc: char;
    lst: lstring (8);
begin
    esc:=chr(27);    write (chr (12) );
    write(esc, '[ 6; 15H', esc, '[ 4 m', '春        晓', esc, '[0m');
    write(esc, '[ 8; 12H', esc, '[ 0 m', '春 眠 不 觉 晓,', esc, '[0m');
    write(esc, '[10; 12H', esc, '[ 1 m', '处 处 闻 啼 鸟.', esc, '[0m');
    write(esc, '[12; 12H', esc, '[ 5 m', '夜 来 风 雨 声,', esc, '[0m');
    write(esc, '[14; 12H', esc, '[ 7 m', '花 落 知 多 少?', esc, '[0m');
    write(esc, '[16; 12H', esc, '[ 8 m', '作者: 孟浩然', esc, '[0m');

```

```

1: writeln;      writeln;
   write ('该诗的作者是谁? :') ;      readln (lst) ;
   if lst='孟浩然' then writeln ('    回答正确',CHR(7))
       else writeln ('    回答错误',CHR(7)) ;

   write('作者是哪个朝代的人? :') ; readln (lst)
   if lst='唐朝' then writeln('    回答正确', CHR(7))
       else writeln('    回答错误', CHR(7));
   write ('还要修改吗? Y或y: 要') ; readln (lst) ;
   if (lst='y') or (lst='Y')
       then goto 1
       else write (esc, '[ 2J')
end.

```

这个程序实现的功能是容易看清楚。运行一开始，先清除屏幕上原来显示的全部内容，并使光标回到屏幕最左上角位置。接下来在 6 到 16 行上，用不同的显示特性，隔行显示唐朝诗人孟浩然的一首诗的题目、四句内容和作者名字。再接下来，向用户提问该诗的作者和生活的朝代，并按用户的回答，指明回答是否正确。最后问用户是否要修改自己的回答，并进行相应的处理，即用户要修改自己的回答时，返回去重新提问，否则，执行清屏操作并结束该程序。

这个程序中，先说明了一个名字为 ESC 的字符型变量，并用 CHR 函数把键盘上的 ESC 键的编码赋给它。这样做，对下面每个用到 ESC 控制串的执行语句是很有用处的。

该程序的执行部分，分为两个以不同方式执行输出操作的部分。前一个部分，用指定输出位置和显示特性的方式输出诗的题目和四行内容。在输出内容的前边给出的显示特性，适用于该项输出的内容。例如，诗的标题是以带下横线方式输出，诗的四行内容则分别以正常特性、高亮度、闪烁和反底色四种不同特性显示。而在第 16 行上的输出，则选用了隐藏方式，所以被输出的内容不会显示在屏幕上。每一项输出内容之后给出的特性控制，用于恢复终端的正常显示特性，目的是保证后续的输出内容按正常特性显示。例如第 16 行上输出的最后两个减号就属于这种情况。用户在设计自己的程序时，一定要注意给出的显示特性起作用的范围。后一个部分，则采用不具体指定输出位置和显示特性的输出方式，这是进行 PASCAL 语言程序设计时更常用的办法。这两种用法同时出现在一个程序中，是我们有意安排的，目的在于比较它们的使用效果上的差别。当该程序开始运行时，诗的名字和内容总是显示在屏幕的同一位置，不管该程序重新运行多少次都将如此。这是因为程序的前一部分明确指明了每项内容在屏幕上的准确位置。而第二部分则不然。例如，用户用字母 'Y' 回答要修改，则程序将再次提出两个问题，并且要用户回答。这第二次的问题将接在前一次问题的下边显示。当用到屏幕最下边一行时，再要用下一行，就会使已显示在屏幕上的内容向上“卷动”。多次卷动后，甚至会把手诗的内容都移没了。这是因为我们把语句标号定到了第二个输出部分的开始处的缘故。不指定输出位置的 WRITE 和 READ 语句在屏幕最底一行遇到回车时，总要引起向上方卷动屏幕内容的操作。但是，当我们把这个程序的语句标号 1 定在程序开始位置时，还存在“卷动”问题吗？不再存在了。请用户思索运行该程序，以加深对此问题的体会。其实，我们也可以在输出“该诗的作者是谁”的语

句中明确指明该内容的显示位置，而不去改动语句标号的位置。看来这后一种方法较好。

该程序在开始处和结束处都执行了一次清屏操作。开始处用的是清屏控制码，结束处用的是 ESC 清屏控制串，二者的执行效果完全相同。

响铃，是用于在 WRITE 语句中输出一个 CHR(7) 的控制码字符实现的。通常可以说一个名字为 BELL 的字符类型的变量，并把 CHR(7) 的值赋给它，后面便可以在 WRITE 语句中使用 BELL 变量，以完成响铃。

给出的第二个程序，是用于完成工资管理中的数据录入功能。程序清单如下：

```
program gzgl (input, output);
label loop;
type
  salary = record
    numb: word;
    name: lstring (20);
    shouru: record
      gb, gl, zw, jj, jb, xl: real
    end;
    zhichu: record
      fz, sf, df, tr, aa, hf: real
    end;
    shifa: real
  end;
var f: file of salary; ch, esc, bell: char;
    person: salary;
    row, col, attr, i, j, l: integer;
    procedure normal;
begin write (esc, ' [0m') end;
    procedure position (row, col, attr: integer);
var r1, c1: lstring (2);
begin
  if (row >= 1) and (row <= 24) and then encode (r1, row: 2)
  then
    if (col >= 1) and (col <= 80) and then encode (c1, col: 2)
    then
      if (attr > -1) and (attr < 9)
      then write (esc, ' [' , r1, ' , ' , c1, ' H' , esc, ' [' , chr (48 + attr), ' m' )
      else write (esc, ' [24; 50H' , esc, ' [5m' , ' 显示属性值错误' )
      else write (esc, ' [24; 50H' , esc, ' [5m' , ' 列值错误' )
      else write (esc, ' [24; 50H' , esc, ' [5m' , ' 行值错误' )
    end;
begin
```

```

esc :=chr (27) ;          bell :=chr (07) ;
assign (f,'salary. dat') ; rewrite (f) ;

```

```

loop: write (chr (12) ) ;
      position (2, 22, 1) ; write ('工资录入操作') ; normal;
      with person do
        [position (4, 08, 0) ; write ('姓名 : ') ;
          position (4, 20, 1) ; readln (name) ; normal;
          position (4, 40, 0) ; write ('职工编号:') ;
          position (4, 62, 1) ; readln (numb) ; normal;

```

```

      with shouru do
        [position (6, 06, 0) ; write ('基本工资 : ') ;
          position (6, 30, 1) ; readln (gb) ; normal;
          position (6, 50, 0) ; write ('工龄工资 : ') ;
          position (6, 70, 1) ; readln (gl) ; normal;
          position (7, 06, 0) ; write ('职务工资 : ') ;
          position (7, 30, 1) ; readln (zw) ; normal;
          position (7, 50, 0) ; write ('奖金数 : ') ;
          position (7, 70, 1) ; readln (jj) ; normal;
          position (08, 06, 0) ; write ('加班费 : ') ;
          position (08, 30, 1) ; readln (jb) ; normal;
          position (08, 50, 0) ; write ('处理费 : ') ;
          position (08, 70, 1) ; readln (xl) ; normal ]

```

```

      with zhichu do
        [position (12,06, 0) ; write ('房租 : ') ;
          position (12,30, 1) ; readln (fz) ; normal;
          position (12,50, 0) ; write ('水费 : ') ;
          position (12,70, 1) ; readln (sf) ; normal;
          position (13,06, 0) ; write ('电费 : ') ;
          position (13,30, 1) ; readln (df) ; normal;
          position (13,50, 0) ; write ('托儿费 : ') ;
          position (13,70, 1) ; readln (tr) ; normal;
          position (14,06, 0) ; write ('缺勤扣除数 : ') ;
          position (14,30, 1) ; readln (aa) ; normal;
          position (14,50, 0) ; write ('工会会费 : ') ;
          position (14,70, 1) ; readln (bf) ; normal ]

```

```

];

```

```

      with person, shouru, zhichu do
        shoua :=gb+gl+zw+jj+xl+jb

```

```

    = fz - sf - df - tr - aa - hf;
    position (17, 30, 7);
    write ('实发数 : ' person. shifa:7:2, '元'); normal;
    f^:=person; put (f);
    position (22, 4, 5);
    write ('还继续录入下一个人的工资? '); normal;
    readln (ch); if (ch='Y') or (ch='y') then goto loop;
    close (f) ; write (chr (12))

```

end.

在这个程序中，首先定义了表示某名职工工资信息的一个记录类型。该记录将由职工编号、姓名、收入情况（由基本工资、工龄工资、职务工资、加班费和洗理费组成）、支出情况（由房租、水费、电费、托儿费、缺勤扣除数和工会会费组成）和工资实发数诸项组成。职工编号与姓名用于标识职工身份，实发数应为各项收入减去全部支出。

我们用给出的记录类型说明了一个文件，用此文件记载全体职工的工资信息。又说明了记录型变量，用于从屏幕上接收工资信息。

我们给出这个程序的主要目的，还是为了表明使用终端控制特性的格式和效果。因此，用户不要急于把注意力放到工资到底应由哪些部分组成，怎样使用记载职工工资信息的文件等方面。

从使用终端屏幕方面看，该程序每次要输入一名职工工资信息时，先是清屏，给出该屏上所执行的操作的标题说明，然后则是顺序地按给出一项提示，进行一次数据录入的方式录入一名职工工资的全部有关数据。最后，求出实发数并把它显示在屏幕上，再把刚录入的内容写到文件中。至此，一名职工的工资信息录入完毕。这种录入方式有无矛盾呢？从每项提示信息 and 读入的数据来看，都通过POSITION过程指定了屏幕位置和显示的特性，计算是方便了。但在运行过程中，还存在另外几个很尖锐的矛盾。第一就是在录入一项数据的过程中，一旦按了回车键，发现输入的数据错了，也已无法改正。有人会说，每个数据记录录入完毕之后，再问一下是否要修改，并按用户回答返回前边，就能解决这个问题。这也可以算一种可行方案，但不大好，实际上是要重新输入一遍数据，包括那些无错的数据项在内，比较麻烦。第二个突出矛盾是，开始的清屏操作之后，提示信息和数据录入是一项一项地显示在屏幕上的，在进行当前一项数据输入的过程中，不知后面还有些什么操作，操作的直观性较差。这是可以解决的。例如，在清屏操作之后，先连续地把本屏全部提示信息都输出完毕，再进行连续地数据读入，这个矛盾就算解决了。但这仍不是最理想的方案。第三个突出矛盾是，当录入的数据中出现非法字符时，将造成该程序运行过程的夭折。如录入基本工资时，输入成A40，该程序会立刻异常结束，屏幕上显示出：

```
? Error: Data format in file USER
```

```
PC=000A:6FD4, SS=6FD4, BP=0FD4, SP=F5F8
```

其后果是，前面已录入的内容完全报废，因为程序结束时，已打开的数据文件尚未关闭，写在文件中的内容丢失了。

要想使这个程序达到可靠实用的标准，必须很好地解决上述三个问题。

录入完一个人的工资信息之后，用户可以回答是否还继续录入下一个人的情况。要继续时，则返回前边，清屏后继续录入下一个人的工资信息。必须指出，每录入一个人的工

资，都得先清一次屏，之后，把某些数据(如提示信息)在相同的位置、用相同的显示特性再重新显示一遍，这是一种不必要的动作。也可不清屏，提示信息保留在屏幕上，这是我们期望的。但有关上一个人的工资信息也同时保留在屏幕上，却是我们不情愿的。如果每录入一个人的工资情况之后，提示信息保持在屏幕上不变，而把要录入数据的位置上的原有字符都清除了，对录入新的数据就方便多了。后面我们将看到解决这个问题的办法。

第三个程序，是个小的示意性的全屏幕编辑程序。所谓全屏幕编辑，通常是指用户在输入或修改被编辑文件内容的过程中，可以随时把光标移到屏幕的任何一个位置，在光标位置修改、插入或删除一些内容(可以是1到多个字符，或1到多行字符等等)，并使这些操作的结果立即显示在屏幕上。采用这种编辑程序编辑文件时，操作简便、直观，易学易用，是目前各类计算机系统中普遍采用的技术。实现这类编辑程序，主要涉及到两项技术。从硬件上讲，要能具体准确地了解所用的终端硬件特性，对PC机来说，就是我们前面介绍的内容以及键盘上每个键的编码。从软件上讲，要会使用“单字符”入/出功能。从程序设计的角度讲，主要是要把屏幕上显示的内容，与程序中被编辑文件的内容在内存中的具体位置准确地对应起来。也就是说，对每一个编辑动作，一方面要把其效果在终端屏幕上表现清楚，另一方面，还必须相应地修改被编辑文件在内存中的实际内容，这二者必须协调好。屏幕上显示的，要正确地体现某片内存区的实际内容。

为了简便，我们的程序中只开了一个24×80个字符型的紧缩数组，让它正好对应终端一个屏幕所能显示的内容。这样，光标在屏幕上的位置(行和列)就可以直接作为访问该数组时用到的下标值。其实，把数组开得更大些，例如开成32kB的一维数组，要解决光标位置上的字符对应这个数组中的哪一个字符，也不是太难的问题，对此我们不再作更多的说明。

实现屏幕编辑程序时，用到的终端控制串主要包括：

- 设置或清除终端状态。
- 移动光标和光标定位。
- 测试光标在屏幕上的当前位置。
- 字符插入、修改和删除。
- 行插入和删除。
- 控制屏幕上的卷动特性。

在这些命令串中，对于测光标位置一项，PC机终端没有用命令串形式提供。用户可以用DOS操作系统提供的中断命令，或直接从操作系统确定的地址中找出光标在屏幕上的行、列位置值。这些内容我们已在程序清单中标明了。

下面是这个程序的清单。

```
program sedt (input, output) ;
type arrc=packed array [1..24] of lstring (80) ;
var  f      : text      ;
     arr    : arrc      ;      emptyline : boolean;
     k      : word      ;      t, tt: byte;
     esc    : char      ;      r, c, first : integer;
     function dosxqq (x, y: word) : byte; extern;
                                     (* 这个函数用于实现单个字符读入 *)
     procedure position (r, c: integer) ;
```

```

begin write (esc, ' [', r:2, ', ', c:2, 'H') end ;
  procedure cursorpos (var row, column: integer) ;
var ar: ads of byte;      (* 该过程用于测出光标当前位置 *)
begin
  ar.s := 16 # 0544;
  ar.r := 16 # 4353; row := ord (ar^+1) ;
  ar.r := ar.r + 1; column := ord (ar^+1)
end;
begin
  esc := chr (27) ; write (chr (12) ) ;
  position (25, 6) ;
  write (esc, ' [7m', '      pf1:结束编辑, 存盘',
        '      pf16:结束编辑, 不存盘      ', esc, ' [0 m') ;
  position (1, 1) ;
  for r:=1 to 24 do
    [arr [r] .len := 0;
    for c:=1 to 80 do arr [r, c] := ' '];
  repeat k:=0;
  tt:=t ; cursorpos (r, c) ;
  repeat t:=dosxqq (6,255) until t < > 0
    if t< >31
    then
      if (t=32) and (tt<30)
      then write (esc, ' [C')
      else
        if t=13
        then write (chr (13) , chr (10) )
        else [arr [r] [c] :=chr (t) ; write (chr (t) ) ;
              if arr [r] .len< wrd (c) then arr [r] .len := wrd (c) ]
    else
      [t:=dosxqq ( 6 , 255) ;
      if t=195 then write (esc, ' [D') else
      if t=193 then write (esc, ' [C') else
      if t=192 then write (esc, ' [A') else
      if t=194 then write (esc, ' [B') else
      if t=199 then (* 字符删除 *) else
      if (t>=128) and (t<=159)
      then [k:=t-127;
            if (k< >1) and (k< >16)
            then write (chr (7) ) ]
      ]
  until (k=1) or (k=16) ;
  if k=1 then
    [assign (f, 'sedt.dat') ; rewrite (f) ;
    emptyline :=false;

```

```

for r:=1 to 24 do
  if arr[r].len=0
    then [if not emptyline then first :=r ;
          emptyline :=true]
    else [if emptyline
          then [for c:=first to r-1 do writeln (f) ;
                emptyline :=false] ;
          writeln (f, arr[r] ) ] ;
  close (f)
]
end.

```

在这个程序中，首先给出了实现单字符读入的一个外部函数说明。这个函数是PASCAL编译程序提供的调用系统子程序的一种手段，在程序中，我们是用 $t:=DOSXQQ(6, 255)$ 方式引用它的，它实现的是把用户在键盘上刚输入的一个字符的编码值以字节类型传送到字节型变量 t 中。用到的两个实际参数6和255表明此时函数完成的功能是单个字符读入。所谓单字符读入，是指用户从键盘上打入的每个字符，都按每打入一个字符，用户程序就立即接收到一个的方式进行工作的。而PASCAL语言中通常的READ语句，则是采用数据缓冲区的方式工作的，即只在用户按了回车键之后，程序才会接收到刚输入的处在同一个输入行上的内容。显而易见，全屏幕编辑程序的字符读入只能以单字符方式进行。例如，用户按了键盘上的 \uparrow 键，编辑程序必须立即做出光标移到上一行的反应，而不能等再打入一个回车键才接收打入的内容。单字符读入方式贯穿于整个程序，对接收到的每一个字符立即进行分析，并依据输入键的不同类别做出不同的响应处理。

DOSXQQ函数的另一项功能，是快速完成字符的显示输出。例如，可以用 $t:=DOSXQQ(6, t)$ 的方式，把字节变量 t 的内容作为一个字符显示到终端屏幕上。从功能上讲，它与WRITE(CHR(t))语句完全相同，只是执行速度要快一些。由此可见，DOSXQQ既用来完成字符读入，也用于完成字符输出，它是通过函数引用时的第二个实参的值是否为255来决定的。不为255时，把第二个实参内容作为字符编码，把该字符输出到终端屏幕上光标的当前位置，并且对 $t:=DOSXQQ(6, t)$ 中的 t 不产生影响。为255时，如 $t:=DOSXQQ(6, 255)$ ，则把从键盘打入的一个字符的编码值送给变量 t 。这里会遇到一个问题，就是当程序执行到该语句时，可能用户已打入了一个字符，此时，把打入的字符接收到变量 t 中就行了。也可能用户尚无打入字符，该语句就不能正确执行。PASCAL的编译程序是这样解决这个矛盾的，当执行该语句时，遇到用户尚无打入字符，即无字符可读时，将把一个零值返回变量 t 中。用户必须在自己的程序中检查变量 t 的内容，以区分是否读到一字符。通常的用法是：

```
REPEAT  t:=DOSXQQ(6,255) UNTIL t<>0;
```

程序中给出的两个过程，一个用于定位光标到屏幕上指定位置，另一个用于读来光标当前位置的行列值。这第二个过程的实现方法，对WANG PC机和IBM PC机是不同的。上面给出的是在王安微机上用到的办法，是直接从指定的内存单元中读得光标所在的行、列值。因为我们目前所用的WANG PC机系统，不允许用户使用小于十六进制的20以下的中断。而在IBM PC机系统中，可以用汇编语言调用INT 10，并通过选用功能码为2

和 8 来实现定位光标和测定光标位置的功能。此时，必须用汇编语言写出实现这个功能的子程序，并在PASCAL的程序中调用它。

在351页给出了在IBM PC机中，执行测定光标当前位置的汇编语言的子程序。用户可以在自己的程序中，把它们说明为外部过程来调用。

现在再回过头来讨论给出的全屏幕编辑程序的主程序部分。先综述程序功能，同时引出实现全屏幕编辑程序用到的有关技术。

首先，清屏后在屏幕的第25行上显示操作提示信息。在真正实用的编辑程序中，编辑操作命令可能很多，提示信息要更多一些，写的可以更简明些。在我们这个小例子中，只示意性的标明可以用功能键 PF1和PF16结束一次编辑过程，PF1 把编辑的结果写入磁盘文件中，PF 16废弃本次编辑的全部内容。

接下来，对编辑程序的数据缓冲区清置为空格，并标明每行中都尚无内容。这里用的缓冲区，被说明为24个LSTRING(80)组成的数组，会对文件的读写操作带来很大的方便。缓冲区开多大，即支持多少行（这里为24），每行的最大长度（这里为80），可以按实际需要确定，当行数和行长比这里给出的值更大时，一屏上就不能显示整个缓冲区的内容，建立屏幕上的内容与缓冲区位置的对应关系，确定光标位置与缓冲区某个字符存储单元的对应关系变得稍为复杂，对编辑程序的实现原理并无影响。

程序的第三部分，是用一个REPEAT语句实现的屏幕编辑的核心内容。我们将比较详细地介绍这些内容，分为四个方面进行讲解：

第一，有关键盘上每个键的编码知识。大家知道，0～31这些编码为终端的控制码，编码0不对应任何一个键，编码1～26对应的是由[CTRL]键分别复合上A～Z中的一个键，编码27对应的是[ESC]键。除此之外，与我们程序有关的还有编码31，它被用于给出移动光标的四个键，字符删除键，32个功能键和其它一些键的编码序列的标记。如[↑]键的编码为31和192两个编码值组成的序列，32个功能键的编码为31，分别和128～159中的一个编码组成的序列等等。编码32～126分别对应终端键盘的一个键，它们才是我们要编辑的真正的字符。了解编码内容，是实现屏幕编辑程序的一个前提。

第二，按读来的每个编码值的具体内容，区分不同情况，做出不同的响应处理，是编辑程序的实质性功能。从大的方面看，可以把编码值分为两大类：一类是用于实现编辑控制的，另一类是编辑输入的真正字符内容。前者用于修改屏幕上显示的内容（包括光标位置）和数据缓冲区的内容，这些控制码本身不被记录到数据缓冲区中。后者则必须被写入到缓冲区中，并同时显示在屏幕上。在我们这个程序中，控制码对应的是由编码31开头的编辑专用键和回车键（编码值为13）组成，被编辑的字符由编码为32的空格和编码33～126的94个可打印字符组成，以及图形符号和汉字等由两个字节表示的符号。在处理被编辑的内容时，关键技术是解决屏幕上显示的内容与编辑的内存缓冲区中的内容的对应关系，这是通过了解光标位置完成的。为此，每读入一个字符，都要先得到光标当前的位置，用的是cursorpos(r, c)语句。由于我们把数据缓冲区开成24行×80列字符大小，与屏幕上实际能显示的内容完全一致，因此，把读来的字符写到缓冲区arr[r][c]（与aarr[r, c]表示法效能完全相同）单元中即可。

第三，控制码的识别及处理问题。在我们的程序中，能识别的控制码是回车码和编辑命令码（由编码31开头）。对回车键的处理是把回车控制码13和进行控制码发送给终端，使光标进到下一行的行首。对编码31的处理是再去读它的后续编码，以便完成可能遇到的

光标↑↓→←四个方向移动的动作，或字符删除的操作，或接收一个功能键的编码值。这些在程序清单中可以看得很清楚。功能键的编码值减掉127就成为功能键的编号1~32。实用中，我们可能只用一部分而不是全部的功能键，此时应检查所按的功能键是否合法，不合法时，响铃以示拒绝，不进行其它处理。这里已给出拒绝接收输入的非法命令的处理方式的例子。结束编辑过程，通常用编辑功能键表示。例如，在我们的程序中，结束REPEAT语句用的条件是UNTIL (K=1) OR (K=16)，反映的就是用PF1或PF16结束本次编辑的操作意图。

第四，被编辑内容的保存与后备版本的处理。在进行一次成功的或不成功的编辑操作之后，可以选择保存本次编辑内容到磁盘文件中，或废弃它们而直接结束编辑程序的运行过程两种不同的处理方案，这在上边的程序中表示得很清楚。与此有关的，我们还想说明另外两点：

(1) 我们的程序，执行的是新建一个被编辑文件的全部内容的操作，而且用的文件名也是硬性指定的 SEDT.DAT。而在实用中，编辑程序用到的数据文件名应由用户给出，并且，不是每次都要重新输入一个被编辑文件的全部内容，还经常要执行修改已有文件内容（包括改变、插入或删除一些内容）的编辑操作。为此，从原理上讲，编辑程序通常要用到两个文件，一个是输入文件，用来读入开始编辑时的初始内容；另一个是输出文件，用来输出本次编辑操作得到的结果。

(2) 在使用编辑程序的时候，我们总是只给出一个数据文件名，既用来指定输入，又用来指定输出文件的文件名，而且还要求本次编辑操作结束之后，原来的输入文件和新建的输出文件同时保留在磁盘上。正常情况下，后者是我们所要的编辑操作的结果，而前者被用作为一个后备版本。为解决文件重名的矛盾，编辑程序要对原来的输入文件进行一次改名操作，使它的文件名中的扩展名部分变为'BAK'。这样，对同一个文件进行多少次编辑，磁盘上也只保留本次编辑之前的唯一一个后备文件版本。

在给出的程序中，保存被编辑的内容是在程序的最后一段实现的。把数据缓冲区说明成行数乘上用LSTRING表示的行长的数组，对TEXT文件的读写是非常方便的。这里要解决两个问题。一是回车键输入，按理说应该属于被编辑内容的正常输入，以表示本行的结束，我们在前边把它完全当作控制码处理了。二是允许上下左右随意移动光标，并在光标位置上显示输入的字符。所以，在向磁盘写入被编辑的内容时，必须把什么内容也没输入的行写成空行，这是用WRITELN (F) 语句实现的。

至此，对本程序的解释全部结束。这个程序的功能非常简单，对大多数读者来说，完全能看懂它。这个程序的许多内容，将要反复用在下一节的程序例子中。

第四个程序例子，是在IBM PC机上实现的。其功能与第三个程序的功能完全相同。程序实现中的主要差别表现在：

(1) 定位光标的过程CURSORPOS和读键盘上每个键输入码的函数KEYIN，都是用汇编语言实现的，被作为外部过程和外部函数用在该程序中。IBM PC机的BIOS软件给出了很多可以在汇编程序中的直接调用的功能调用，以支持各种基本的输入输出操作功能，其中包括定位光标，测定光标当前位置、读键盘等等功能。

(2) 键盘上每个键的编码方案和某些键的具体编码值不完全相同。在IBM PC机中，每个键都有两种编码，即扫描码和ASCII编码。每个键都有唯一的扫描码，其编码值是按键在键盘上的位置排定的。ASCII编码只适用于95个可打印字符，和1~31这些控制

码。编辑专用键和功能键等的ASCII编码值均为零。

下面是该程序的程序清单。

```
program sedt (input, output);
type arr:=packed array [1..24] of string (80);
var f : text;
    arr:arr; emptyline:boolean;
    k, w:word; scan, asc:byte;
    esc:char; r, c, first:integer;
    procedure cursorpos (var y, x:integer); extern;
    function keyin:word; extern;
    procedure position (r, c:integer);
begin write (esc,'[',chr(48+r div 10),chr(48+r mod 10),',',
            chr(48+c div 10),chr(48+c mod 10),'H')
end;
    procedure readcode;
begin (* 读来一个键的扫描码和ASCII码 *)
    w:=keyin; scan:=hibyte(w);
    asc:=lobyte(w)
end;
begin
    esc:=chr(27); write (esc,'[J');
    position (22, 6);
    write (esc,'[7m','pfl:结束编辑, 存盘,
            pfl6:结束编辑, 不存盘', esc,'[0m');
    position (1, 1);
    for r:=1 to 24 do
        [arr[r].len:=0;
        for c:=1 to 80 do arr[r, c]:=' '];
repeat
    [ cursorpos.(r, c);
    readcode;
    if (asc=13) and (scan=28) then write (chr(13), esc,'[B')
    else
        if asc=0 then
            if scan=72 then write(esc,'[A') else
            if scan=80 then write(esc,'[B') else
            if scan=75 then write(esc,'[D') else
            if scan=77 then write(esc,'[C') else
            if (scan=59) or (scan=64) then k:=scan-58
            else write (chr(7))
```

```

        else
        [arr [r] [c] :=chr (asc) ; write (chr(asc)) ;
        if arr [r] . len < wrd(c) then arr [r] . len := wrd(c)]
    ]
until (k=1) or (k=6) ;
if k=1 then
    [assign (f,'sedt.dat'); rewrite (f) ;
    emptyline :=false;
    for r:=1 to 24 do
        if arr [r] . len=0
            then [if not emptyline then first:= r ,
                emptyline:=true]
            else [if emptyline
                then [for c:=first to r-1 do writeln (f) ;
                    emptyline :=false] ;
                writeln (f,arr [r] ) ] ;
    close (f)
]
end.

```

第五个程序，是终端处在插入状态下执行输入的一个例子。

在使用编辑程序的过程中，每个人都用到过编辑程序的替换与插入两种输入方式。在替换方式下，从键盘打入的字符，将简单地替换掉光标位置上的原有字符；在插入状态下，从键盘打入的字符，将出现在光标当前位置，而原来处在光标位置上的字符和在同一行中跟在它右边的全部字符，都将右移一个字符位置。例如，光标所在行内容为 A B 1 2 3，光标在字符'1'处，当从键盘输入一个字符C时，在替换方式下，该行的内容变为 ABC 2 3，在插入方式下，它将变为 ABC 1 2 3。这两种编辑方式是可以相互切换的。

终端是通过两种不同的工作状态，来支持上述两种输入方式的。当终端处于插入状态时，它支持插入方式的输入操作。用户可以通过按 INSERT 键来控制终端进入或退出插入状态。当终端处在非插入状态时，按 INSERT 键，将使终端进入插入状态，并用点燃键盘上方相应的一个指示灯来表明这种状态，这之后再按 INSERT 键，将使终端退出插入状态，并用熄灭相应指示灯来表明此种状态。

这里给出的程序，就是用来表明上述操作过程的一个例子。下面是这个程序的程序清单。

```

program sedt (input, output) ;
var lstr: lstring (20) ;
    x, y: word ; c: byte;
    ins: boolean ; ch, esc, bell: char;
    j :integer;
function dosxqq (x, y:word) : byte; extern;
begin

```

```

write (chr (12) ) , ins:=false;
bell:=chr (7) , esc:=chr (27) ;
write (esc, ' [8;20H' ) ;
lst1 := '1234567890abcdefghijklmnopqrstuvwxyz';
write (lst1, esc, ' [8;20H' ) , j:=0;
repeat
  c:=dosxqq (6,255) ;
  while c=0 do c:=dosxqq (6,255) ;
  if c=31
    then
      [c:=dosxqq (6,255) ;
      if c=198
        then
          [ins:=not ins;
          if ins then ch:='h' else ch:='l';
          write (esc, ' [11;4',ch) ]
        else write (bell)
      ]
    else
      [if not ins
      then write (chr (c) )
      else
        write ('? ',esc,' [4l',esc,' [D',
          chr (c) ,esc,' [4h' ) ;
        j:=j+1; lst1 [j] :=chr (c) ]
      until c=wrđ ('? '); write(esc,' [11;4l' ) ;
      write (esc,' [10;20H',lst1)
end.

```

end.

这个程序的功能，是先在屏幕的第8行第20列的位置开始，显示LST1变量的内容，然后用替换或插入方式，读入若干字符，在遇到字符‘?’时结束输入。

在这个程序中，通过检查读来的字节的值是否为31，来区分是控制串还是正常的输入内容。不为31，就是正常的输入内容。在把它显示到终端屏幕的同时，还把它赋给LST1的某个单元。在终端状态不同时，显示读来的内容的办法也不相同。在非插入状态下，直接输出就可以了，而在插入状态下，就要复杂一些，正如程序中给出的语句所表示的。若输入的一个字节值为31，表明给出的是控制串，即用户按的是编辑功能键。当跟在后面的一个字节值为198时，表明按的是INSERT键，不为198时，在我们的程序中作为出错进行处理，就响铃以表示拒绝接收该键输入。在读到一次INSERT键输入时，使我们程序中的变量ins取反，并按它的值设置或清除终端插入状态，同时点燃或熄灭相应的指示灯。这是用命令串ESC[11;4^h来实现的，4h表示设置终端为插入状态，4l表示退出插入状态，11h表示点燃指示灯，11l表示熄灭指示灯。这在介绍终端硬件特性时已讲到过。

这个程序结束之前，应恢复终端状态为非插入状态（标准设置状态）。

这个程序的最后一个语句，输出LST1的内容。读者很容易发现，在执行过插入方式的输入操作后，LST1的内容和在第8行上显示的内容（前20个字符）不吻合。这就是我们在前边的几个程序中，一再提到的必须使屏幕上显示的内容和内存缓冲区中的内容很好地对应的含义。这里的运行结果，是二者没能正确的对应起来的具体体现。很容易从程序中发现这个错误。在插入方式下，屏幕上的内容是按插入方式处理了，但对LST1的内容却仍是按替换办法处理的。此时，可对该程序做如下修改：

- 在程序的说明部分再多说明一个整型变量 i；
- 在语句 $j := j + 1$ 之后，增加如下一个语句：
for $i := 20$ downto $j + 1$ do $lst1[i] := lst1[i-1]$ ；

修改后的程序，运行结果是正确的。

如果读者有兴趣，可以把这个程序的有关内容和思路，加入到第三个程序中去，使那个小的编辑程序，还可以在替换与插入两种方式下运行。

6.2 实现屏幕格式数据输入输出的基本技术

在第6.1节中，我们对屏幕格式数据的输入输出提出了种种要求，提到了许多要解决的具体问题。我们将在本节给出解决这些问题的基本技术和有关思路。下面从四个方面进行说明。

(1) 从支持屏幕格式数据入/出的角度讲，应该把有关的支持软件做成PASCAL语言的一个MODULE或UNIT，供所有用户方便调用。这就等于扩充了原有PASCAL语言的基本功能，增加了屏幕上进行格式数据入/出的新的语句。

(2) 从程序设计的角度讲，最好能把支持软件以结构性能良好，功能层次清晰的语句方式提供给用户。也就是说，能用若干个语句，把一屏数据入/出的全部要求表述清楚。例如：

ACCEPT (i)；

AT (x, y, z, ...)； AT (.....)；

.....

KEYS (p, q, r...)

用ACCEPT表示执行一屏数据入/出的一组语句的开始，用它的实参表明对此屏输入输出操作的特定要求，例如，是否要先清除输入变量原有值之后再执行输入。

用KEYS表示执行一屏数据入/出的一组语句的结束，用它的实参表明对此屏输入输出的另外一些特定要求，例如，本屏操作时可用的功能键和是否允许执行输入等。

用夹在ACCEPT和KEYS之间的每一个AT语句，表示对本屏中输入或输出的一个具体数据项的全部要求，例如，是输入还是输出，入/出的数据的变量名和类型是什么，入/出在屏幕上的准确位置，入/出的字符个数，及以显示入/出字符所使用的显示特性等等。

这样处理的好处是明显的。一是对一屏数据入/出的全部要求，能分层次的表述清楚；二是一屏数据入/出的全部语句结构清晰，书写阅读直观方便；三是能方便地对本屏数据的全部输入采用类似于全屏幕编辑程序的方式进行处理。

(3) 从程序运行的角度讲，最好把一屏上全部的输出信息，能一次完整地显示到屏幕上，而不管这些输出语句之间有还是没有输入语句，有一个还是多个。对所有的输入语句，要能按任意次序、而不是严格的按它们在程序中的书写次序来执行输入操作。这是这种实现方案中最难解决的技术难题。在执行这些输入操作的过程中，既要可以通过把光

标移到某个输入区，向相应变量输入内容，也要可以方便地从一个输入区直接跳到下一个输入区。

(4) 在上述三项基本要求（或者说三项实现技术）之外，还应提供另外一些技术支持，例如，一屏数据入/出的束结方式、重复执行一屏数据入/出操作时的特殊处理等等。

下面我们用MODULE形式给出大体上能满足上述要求的一个支持软件。在清单之后再对程序的有关内容进行详细的说明。

```
( * $warn+ * )  
module imp [public]  
type  
  str=istring (80) ; aad=ads of str;  
  keys= set of 1..10;  
var  
  s: array [0..10] of  
    record  
      attr,lth,row,column:integer;  
      adress:aad  
    end;  
  fir:boolean;  
  first: integer;  
  esc,bell,frame:char;  
value  
  esc:=chr (27) ; bell:=chr (7) ; frame:=chr (135) ;  
  FUNCTION DOSXQQ (A,B:WORD) ;BYTE;EXTERN;  
  procedure CLS;  
begin  
  write (chr (12) ,chr (27) ,' [0m')  
end;  
  procedure ATTRIBUTE (attr:integer) ;  
begin  
  IF attr and 15=0  
  then write (esc,' [0m')  
  else  
    [if attr and 2< > 0 then write (esc,' [5m') ;  
    if attr and 1< > 0 then write (esc,' [1m') ]  
end;  
  procedure POSITION (ro,col:integer) ;  
begin  
  IF (ro in [1..24] ) and (col in [1..80] )  
  then write (esc,' [',ro:2, ',',col:2,'H')  
end;  
  procedure ACCEPT (inlv: integer) ;
```

```

begin
  fir:=initv=0; first:=0;
  s[0].lth:=0
end;

procedure AT (attr1,lth1,row1,column1, integer, ss,aad) ;
var l,i: integer;
    t:byte;
begin
  position (row1,column1) ;
  attribute (attr1) ;
  if attr1 and 16#400<>0
  then [ if first=0 then
        for l:=1 to lth1 do
          if l<=ord (ss^[0]) then
            [ t:=WRD (ss^[1]) ; t:=DOSXQQ (2,t) ]
        ]
    else
      if s[0].lth<10 then
        [ for l:=1 to lth1 do
          [ if fir
            then [ write (frame) , ss^[1] := ' ' ]
            else
              if (ss^[1] <> ' ') and [l<=ord (ss^[0]) ] )
              then write (ss^[1])
              else write (frame)
            ] ,
          if first=0 then [s[0].lth:=s[0].lth+1 ;
            with s[s[0].lth] do
              [ adress:=ss ; attr:=attr1 ; lth:=lth1 ;
                row:=row1 ; column:=column1 ] ]
          ] ,
          attribute (0)
        end ;

        procedure CURSORPOS (var row,column, integer) ,
        var ar, ads of byte ;
        begin
          ar.s:=16#0544 ;
          ar.r:=16#4353 ; row:=ord (ar^+1) ;
          ar.r:=ar.r+1 ; column:=ord (ar^+1)
        end ;

```

```

PROCEDURE KEYS (noinput: integer ; kee: keys ;
                var key: integer ) ;

```

```

VAR

```

```

    J, L, K, P, I: integer;

```

```

    tl, tt    : word    ;

```

```

    ro, co    : integer  ;

```

```

    valid     : boolean  ;

```

```

    function inside: boolean

```

```

    var bl: boolean ;

```

```

begin

```

```

    j := 1 ;

```

```

    inside := false ; bl := false ;

```

```

    while not bl and (j <= s [ 0 ] .lth) do

```

```

        with s [ j ] do

```

```

            if (ro=row) and (co >= column) and (co < column + lth)

```

```

            then [ inside := true; bl:=true;

```

```

                    p := co - column + 1 ]

```

```

            else j := j + 1

```

```

        end;

```

```

    procedure delchar ;

```

```

    var i : integer ;

```

```

begin

```

```

    if NOINPUT = 0 AND then inside then

```

```

        if p <= ord (s [ j ] . adress ^ [ 0 ] )

```

```

        then with s [ j ] do

```

```

            [ delete (adress ^, p, 1) ;

```

```

                write (esc, ' [s', esc, ' [5h' ) ;

```

```

                attribute (attr) ;

```

```

                for i:=p to lth do

```

```

                    if i <= ord (adress ^ [ 0 ] )

```

```

                    then write (adress ^ [ i ] )

```

```

                    else write (frame) ;

```

```

                write (esc, ' [0m', esc, ' [u', esc, ' [5l' )

```

```

            ]

```

```

        else write (bell)

```

```

    else write (bell)

```

```

end ;

```

```

BEGIN      (* keys *)

```

```

    with s [ 0 ] do if lth > 0

```

```

        then [ row:=s [1] .row ; column:=s [1] .column ]

```

```

    else [ row:=1 , column:=1 ] ;
first:=1 ;
position (s [ 0 ] .row,s [ 0 ] .column) ;
repeat
    key := -1 ;
    attribute ( 0 ) ;
    cursorpos (ro,co);
    tt := t1 ;
    repeat t1:=dosxqq (6,255) until t1 ( ) 0 ,

    if t1=9 then
        [ if j<s [ 0 ] .lth
            then j:=j+1 else j:=1 ,
            position (s [ j ] .row,s [ j ] .column) ;
        ] else
    if t1=31 then
        [ t1:=dosxqq (6,255) ,
            if t1=195 then write (esc,' [1D') else
            if t1=193 then write (esc,' [1C') else
            if t1=192 then write (esc,' [1A') else
            if t1=194 then write (esc,' [1B') else
            if t1=199 then [if noinput= 0 delchar else write(bell);] else
            if (t1>=128) and (t1<=137) then
                [ k:=ord (t1) -127 ,
                    if k in kee then key:=k else write (bell) ]
            ] else
    if inside and then noinput= 0 then
        [ case s [ j ] .attr div 16#100 of
            1: valid := (t1=32) or (t1>47) and (t1<58) ,
            2: valid:= (t1>31) ,
            otherwise
        end;

        if valid then
            if (t1=32) and (tt<30)
                then write (esc,' [1C')
            else with s [ j ] do
                [adress^ [p] :=chr (t1) ;
                    t1:=dosxqq ( 2,t1)
                ]
                if adress^ [0] <chr (p)
                    then adress^ [0] :=chr (p)

```

```

        ]
        else write (bell)
    ] else write (bell)
until key < > -1
end,
END.

```

在这个公用的程序模块中，定义了三种数据类型：

STR是最大长度为80的变长字符串类型，AAD为该串的段际地址类型；这两个类型是说明和调用过程AT要用到过程的参数类型。

KEYSS是用于限定所用的功能键的一个集合类型。

对这三个类型，还必须在调用该模块的程序中进行说明，其方式与这里的说明办法相同。

在变量说明部分，说明了一个S数组。它的0号元素用于记录一屏入/出数据中的输入数据项的数目，其它元素用于记录对每个输入数据项的要求的有关信息，这包括：

- 是输入还是输出（此处一定为输入），输入的是什么类型的数据；我们在这里规定数据可以是整型或字符串型两种。

- 用什么显示属性显示输入的字符；我们这里规定可以是正常属性、高亮度或闪烁三种。

我们是用它的ATTR域来记忆上述两部分内容的，并且规定：

（1）用整型变量的高位字节表示入/出以及输入的数据类型，当该字节的值为4时表示输出，为1时表示输入整型，为2时表示输入字符串。

（2）用该整型变量的低位字节表示显示的属性，当该字节的值为0时，表示正常属性，为1时，表示高亮度，为2时，表示闪烁属性。此属性规定亦适用于输出。应该用16进制形式给出该整型量的值。例如，16#100表示的是用正常属性输出有关内容，16#101表示的是用高亮度属性输入整型量，16#202表示用闪烁属性显示输入的字符串等等。用16进制能自然而简便地分清高低位字节各自的值。

- 输入或输出的每个数据项的长度，在屏幕上的行、列位置。这是用LTH, ROW和COLUMN三个域表示的。

- 输入或输出的每个数据项的段际地址。这里是用ADDRESS域表示的。

上述规定对输入与输出都适用。在数组S中只记录对输入数据项的有关规定，没有必要把对输出数据项的规定也记忆下来。对输入而言，不管输入的是什么类型的数据，都只能先把输入的内容以字符串形式读进来，给出的类型，在这里只用于对输入字符的合法性检查。例如，在读入整型量时，输入的字符若不在‘0’..‘9’之内，表明输入的是非法字符，将响铃并拒绝接收。

在变量说明部分，还说明了fir和first两个变量。它们的用法将在有关过程中介绍。

ESC, BELL和FRAME分别是[ESC]键的编码，响铃码和一个小亮方格的图形符号，都被说明为字符类型，并在程序的VALUE说明部分给定了初值。这三个字符变量用在WRITE语句中。

这个模块的核心部分，是给出的若干个过程，它们是其它程序要调用的直接对象。这些过程是：

CLS, ACCEPT, AT和KEYS

该模块内部还说明与使用了函数DOSXQQ、INSIDE和过程ATTRIBUTE、POSITION、CURSORPOS, DELCHAR;

下面对这些函数与过程进行讲解。

DOSXQQ 和 CLS, 用于单字符入/出和完成清屏、恢复终端正常属性, 在前边给出的程序中已经用到过, 此处不再说明。

定位光标 (POSITION) 和测光标位置 (CURSORPOS) 两个过程, 与在前边用过的完全相同。

ATTRIBUTE过程, 变换入/出要求中的属性信息, 为终端硬件所要求的值, 并把一个相应的ESC序列的控制串送往终端。这个过程完成对显示属性的控制, AT过程和KEYS过程都要调用它。

ACCEPT过程的功能是:

- 用它的参数initv, 表明执行本屏的数据输入之前, 是否要首先清所有输入变量为空格之后再执行输入。initv为0, 表明要清; initv为1, 表明不清, 即执行的是修改已有数据。执行过fir:=initv=0语句之后, fir变量记忆的就是这个信息。这个信息将在AT过程中被检查使用。AT过程只能查询fir变量, 而不能直接使用ACCEPT过程的参数initv。

- 记下首次进入一屏数据入/出处理的过程, 用first:=0语句实现。只有在进入KEYS过程后才让first为1。这在连续地多次执行一屏数据入/出处理的情况下, 会得到很明显的好处。进一步的解释将在介绍AT过程时给出。

- 清S的0号元素中的lth域的值为0, 为对执行输入功能的AT过程的数目进行计数做好准备。

AT过程的功能是:

- 对输出数据项而言, AT过程要完成数据的输出操作。AT过程的参数attr1, lth1, row1, column1和SS分别给出的是输出时的显示属性, 输出串长、在屏幕上的行、列位置和输出串的段地址。

执行输出的具体步骤是:

定位光标到指定位置;

给出指定的显示属性控制;

检查是否为输出, 并且是否是首次执行此AT语句 (用判别first是否为0实现)。在条件成立时, 按指定的长度输出有关串变量。从这里可以看到, 当first不为0时, 该过程不执行输出功能。这就是我们前边几次提到的, 重复执行同一屏数据入/出时, 对输出数据只在首次执行时一次输出即可, 后面将只处理输入数据项。同时可以看到, 输出时还可以只输出一个长串的一部分内容, 当给定的长度比实际要输出的串的长度大时, 也只输出该串的实际内容, 不会由于长度给的不合理, 输出错误的内容。

最后一步是恢复终端的正常显示属性。

对输入数据项而言, AT过程只完成布置屏幕格式、对AT过程的数目计数和填写数组S相应元素的内容 (即登记好对本输入项的全部要求) 三项功能, 真正的输入并不在AT过程中完成, 这是这个支持软件的关键技巧。

执行输入的具体步骤是:

定位光标到指定位置;

给出指定的显示属性控制;

判定输入的数据项数目是否超过S数组所能登记的最大数目。不超过时,执行具体的处理过程。也就是,当要先清输入变量为空格再输入时,要先输出指定个数(1th1给定的长度)的亮方块符号到屏幕上,并把空格字符赋给输入串的每一个字符元素。否则,要把输入串的原内容显示到屏幕上来。显示时,正常显示输入串中的非空格字符,把空格字符和尚无内容的部分显示为小亮方块。

如果是首次执行此AT过程,还要完成对AT过程数目的计数功能,和填写S数组的功能。

最后一个步骤,是恢复终端的正常显示属性。

KEYS过程,是这个支持软件的最重要的一个过程。它的功能是完成一屏数据真正的输入操作。它的三个参数的意义是:

NOINPUT表示允不允许执行输入,为0代表允许,为1则不允许。正常的用法,应该选NOINPUT为0值。当只要显示某些数据的值而不允许修改变动这些数值时,只要使NOINPUT由0变为1即可,而不必对处理一屏数据入/出的语句做任何变更。

KEE参数用于给出本屏操作时可用的功能键集合。KEY参数用于记忆接收到的功能键编号。结束一屏数据的输入过程,是通过按选定的任何一个功能键(有时也用回车键)来控制的。这可以通过KEYS过程中的REPEAT...UNTIL KEY< >-1语句看得很清楚。

KEYS的具体执行步骤是:

定位光标到第一个输入数据项的位置,在本屏无输入项时,则定位光标到屏幕最左上角。

执行真正全部输入。这是由一个REPEAT语句完成的。每当读来一个字符后,就按不同情况执行不同的处理:

当读来的字符为TAB键时(即 $t_1=9$),表明希望把光标从一个输入区移入下一个输入区。我们是用变量J来表示输入项的序号的。当J不是最后一个输入项时,就使J的值变为J+1,否则使J为1,然后按S[J]中登记的ROW和COLUMN的值定位光标。

当读来的字符的编码为31时,表明用户按了一个编辑专用键或一个功能键,具体的键由紧跟在其后的另一个编码决定。为此,在接到一个31编码后,要跟着读下一个编码。我们在这个支持软件中,处理了上、下、右、左移光标,字符删除和接收1到10这10个功能键,其它情况均视为非法字符,将响铃并拒绝接收其输入。编码192,194,193和195分别为上下右左移光标的键,用相应的ESC序列控制串执行光标移动。编码199为删除键,转DELCHAR过程执行字符删除处理。编码为128到137表明1到10十个功能键,减127后就是功能键的编号。当该编号属于KEYS过程中的KEE集合时,表明读到一个合法的功能键,否则接收到的是本屏不允许使用的功能键,就响铃并拒绝接收该输入,保持KEY为-1,使REPEAT语句得以继续重复执行。

字符删除是调用DELCHAR过程执行的。在读到删除键时,首先要检查NOINPUT是否为0,只有它为0时,才允许删除字符(即允许输入),否则响铃表示不能执行删除。

删除得以执行的条件是,光标在某个输入区并且光标位置有字符可删。判断光标是否在某个输入区,是通过引用INSIDE函数完成的。当INSIDE为TRUE,表明在某个输入区,为FALSE表明不在任何一个输入区。判断的办法,是把光标当前位置值,与记录在数组S中的每个元素的有关内容比。当INSIDE为真时,还得到了光标所在的输入区的序号值J。INSIDE函数在执行数据输入时,也将被引用。

删除一个字符，要进行两方面的处理。一是要修改输入变量本身的值，是用MS PASCAL语言提供的处理变长字符串的一个过程 DELETE完成的，要给出执行删除串的变量 (address↑)，删除字符的起始位置 (P) 和删除的字符个数 (1)；二是要修改屏幕上显示着的有关内容。我们采用的办法是把相应串修改后的部分内容重新输出一遍（从位置P到串尾），而没有采用终端硬件提供的字符删除命令串。原因是该命令串将使本行中光标位置之后的全部内容左移一个字符位置。如果本行后半行还有其它的输入或输出项，它们也将受到影响，这是不允许的。在处理字符删除的过程中，用到了好几个控制光标特性的控制串，它们的功能可以在本章第一节的内容中找到。

当读出的字符不属于上述两种情况时，则认为正常的字符输入。能执行正常的字符输入的条件，与能执行字符删除的条件是相同的，即NOINPUT的值为0，并且光标在某输入区。在上述条件成立时，还应按输入的数据类型检查输入字符的合法性。在我们的程序中，输入字符串时，合法字符为等于或大于空格的全部字符。输入整型量时，合法字符为‘0’到‘9’，开头或结尾位置也允许输入空格字符。在有效数字符之间出现空格是没有道理的，但程序中未对此进行检查。

当输入的为合法字符时，就要把该字符写入到输入串相应位置，并把它显示到屏幕上，还应按该字符在串中的位置；在需要时修改输入串中实有字符的个数（即串的实际长度）。

当输入的是一个可用的功能键时，KEYS将结束输入过程。

这个支持软件，可以被方便地连接到用户程序中，实现屏幕上格式数据的输入/输出操作。

6.3 在用户程序中使用支持软件

在上一节，我们给出了一个支持屏幕上格式入/出语句的小软件。在本节将给出使用这一支持软件的几个程序例子。

第一个例子，是实现在屏幕上显示一行数字1，在另外三个位置上读入两个整数和一行字符的小程序。可用功能键规定为1~5，用于结束本程序的运行过程。最后还要在屏幕22行的40列的位置开始显示LST4的结果，最后这个语句也可以换成AT (#400, 20, 22, 40, ADS LST4)。

```

program test (input,output);
type str=lstring (80);
    aad=ads of str;
    keyss=set of 1..10;
var
    lst1,lst2,lst3,lst4: str; k: integer;
    procedure cls; extern;
    procedure accept (initv: integer); extern;
    procedure at (attr,lth,row,column: integer; ss:aad); extern;
    procedure keys (noinput: integer; keo:keyss;
                    var key: integer); extern;
begin cls;
```

```

1st1:= '11111111111111111111111111111111'
accept (0) ;
  at (16#400,20,10,20,ads'1') ;
  at (16#101,20,12,20,ads'1') ;
  at (16#201,20,14,20,ads'1') ;
  at (16#101,20,16,20,ads'1') ;
keys (0, [1..5],k) ;
writeln (chr (27) .. [20,10,1,1,1]) ;
end.

```

在运行这个程序时，可反复进行各种试验和检查，例如，在非输入区试着输入字符，用TAB键在各输入区之间跳跃，在某一输入区输入合法的、非法的字符，执行字符删除功能，按动合法的或禁用的功能键等等。还可以变换ACCEPT过程、AT过程和KEY过程的各参数，试看各参数的作用、效果等等。

有关在程序中使用MODULE的问题，已在本书第一章详细介绍过，此处不再说明。要注意，程序和模块中公用的数据类型（主要用于定义过程、函数的参数），在两部分中都要说明，在程序中，还要把程序要用的那些在模块中实现的过程、函数，说明为程序的外部过程和外部函数。

第二个例子，是把本章第一节中给出的程序POET进行改写，用屏幕上格式数据入/出支持软件实现相同功能的一个程序。我们已把公用数据类型的定义和外部过程说明的部分写到一个名字为TYPPROC的TEXT文件中，并用\$INCLUDE编译命令，把该文件“引入”到程序中，这种用法要更方便一些。

下面是该程序的源清单：

```

program boel(input,output) ;
  * $include:'typproc' *
label 1;
var i,j,k:integer ;
  esc: char ;
  lci: lstring(8);
begin
  upcls(1,1,21,80,0);
  accept(0) ;
  at(#40,05,20,1,15,ads'春 晓 ');
  at(#40,05,20,2,15,ads'春 眠 不 觉 晓,');
  at(#40,05,20,3,15,ads'处 处 闻 啼 鸟,');
  at(#40,05,20,4,15,ads'夜 来 风 雨 声,');
  at(#40,05,20,5,15,ads'花 落 知 多 少,');
  at(#40,05,20,6,15,ads'作者: 孟浩然');
  at(#40,05,23,8,15,ads'该诗的作者是谁? ');
  at(#30,05,3,8,46,ads'1st1;

```

```

at(#40,05,33,10,15,ads'作者是哪个朝代的人? :');
at(#30,05, 8,10,46,ads 1st);
at(#40,05,20,18,10,ads 'F1:结束回答 F10:结束运行');
keys(0, [1,10],k,y);

case key of
1: [if 1st='孟浩然' then at(#40,05,13,16,15,ads'回答正确')
    else at(#40,05,13,16,15,ads'回答错误');
    if 1st='唐 朝' then at(#40,05,13,17,15,ads'回答正确')
    else at(#40,05,13,17,15,ads'回答错误')] ;
10: upcls(1,1,21,80,0) end
end.

```

这个程序的结构非常清晰。第一个语句完成清屏，随后ACCEPT和KEYS两个过程，以及它们之间的全部AT过程，控制屏幕上一屏数据的输入和输出显示等。在输入过程中，可以把光标移到允许输入字符的位置并执行输入。当按功能键[F1]时，表示结束回答，按[F10]键时，结束程序的运行过程。在KEYS语句之前的那一个AT语句，用于给出操作提示信息。

程序最后部分的CASE语句，按用户结束屏幕入/出操作所用的功能键，执行不同的处理功能。在我们这个程序中，用户按了[F1]键，就要检查用户回答的正确性，并给出对检查结果的答复，程序随之结束运行，屏幕上已显示的内容不变。用户按了[F10]键，完成清屏后结束程序的运行过程。该程序给出的是使用屏幕格式数据入/出支持软件的通用方式。

第三个例子，是实现工资管理任务中的数据录入功能的一个程序。

程序中说明了一个存放录入结果的记录类文件F，说明了一个个人工资组成情况的记录型变量PERSONSL，这二者之间有直接的对应关系，即F^的每一个域与PERSONSL的每一个域都相同，这样，程序写起来更简便。Salary类型中的全部数值域(除SHIFA域外)都被说明成LSTRING类型，是支持软件所要求的。其实也可以把F^的每个数值域都说明为INTEGER。完成一个人的工资后，必须把接收到的每个整型量的数据串变成二进制形式的整型量，才能求出实发数，所以实发数是计算出来的。最后一个语句用于清屏。

下面是该程序的源清单。

```

program gzgl(input,output);
(*  *include:'interf.pas' *)
type
salary = record
    numb : lstring(5);
    name : lstring(20);
    shou: record
        lgb, lgl, lzw, lj, ljb, lxl: lstring(4)
    end;
end;

```

```

        zhichu: record
            lfz, lsf, ldf, ltr, lqq, lhf: lstring(1)
        end;
        shifa: integer
    end;

var
    f: file of salary;
    personalsal: salary;
    key, gb, gl, zw, jj, jb, xl, fz, sf, df, tr, qq, hf: integer;

begin
    assign(f, 'salary.dat'),      f.mode := direct;
    rewrite(f);
    upcls(1, 1, 21, 80, 0);
    repeat
        accept(0);
        at(#40, 05, 14, 2, 22, ads'工资录入操作'),
        with personalsal, shouru, zhichu do
        [at(#40, 05, 8, 4, 8, ads'姓名'), at(#30, 05, 20, 4, 20, ads name);
        at(#40, 05, 10, 4, 40, ads'职工编号'); at(#10, 05, 5, 4, 62, ads numb);
        at(#40, 05, 10, 6, 6, ads'基本工资'); at(#10, 05, 4, 6, 20, ads lgb);
        at(#40, 05, 10, 6, 40, ads'工龄工资'); at(#10, 05, 4, 6, 62, ads lgl);
        at(#40, 05, 10, 8, 6, ads'职务工资'); at(#10, 05, 4, 8, 20, ads lzw);
        at(#40, 05, 8, 8, 40, ads'奖金数'), at(#10, 05, 4, 8, 62, ads ljj);
        at(#40, 05, 8, 10, 6, ads'加班费'), at(#10, 05, 4, 10, 20, ads ljb);
        at(#40, 05, 8, 10, 40, ads'洗理费'), at(#10, 05, 4, 10, 62, ads lxl);
        at(#40, 05, 8, 12, 6, ads'房租'), at(#10, 05, 4, 12, 22, ads lfz);
        at(#40, 05, 8, 12, 40, ads'水费'), at(#10, 05, 4, 12, 62, ads lsf);
        at(#40, 05, 8, 14, 6, ads'电费'), at(#10, 05, 4, 14, 20, ads ldf);
        at(#40, 05, 8, 14, 40, ads'托儿费'), at(#10, 05, 4, 14, 62, ads ltr);
        at(#40, 05, 10, 16, 6, ads'缺勤扣除'); at(#10, 05, 4, 16, 20, ads lqq);
        at(#40, 05, 10, 16, 6, ads'工会会费'); at(#10, 05, 4, 16, 62, ads lhf);
        at(#40, 05, 8, 18, 30, ads'实发数');
        at(#40, 05, 40, 20, 10, ads'F1: 此人工资录入完毕      F5: 结束程序运行'),
        keys(0, [1, 5], key);
    case key of
        1: [if decode(lgb, gb) and decode(lgl, gl) and decode(lzw, zw) and
            decode(ljj, jj) and decode(ljb, jb) and decode(lxl, xl) and
            decode(lfz, fz) and decode(lsf, sf) and decode(ldf, df) and

```

```

        decode(ltr,tr) and decode(lqq,qq) and decode(lhf,hf)
    then shifa:=(gb+gl+zw+jj+jb+x1)-(fz+sf+df+tr+qq+hf);

    position(18,46); write(personal.shifa:5);
    f:=personal; put(f)
  ];
  5: close(f)
end
]
until key=5;
upcls(1,1,21,80,0)
end.

```

用这种方式录入数据时，屏幕上的格式清晰，录入直观方便，能直接排除录入中的某些错误对程序运行的不良影响，可以在整个屏幕上以类似于全屏幕编辑的方式执行一屏数据的录入过程。而且编写起完成数据录入的主程序来特别简便，的确是一种从终端录入数据的良好手段。

6.4 在屏幕上实现多窗口入出的技术

多窗口入/出技术，是指在一个终端屏幕上，划分出多个不同的区域，供多个不同的程序或进程独立使用。每一个区域被称为一个窗口（WINDOW）。使用多个窗口技术，类似于同时使用个数更多（2个或2个以上）、面积更小（实际终端屏幕的一部分）的多个独立的显示屏幕。多窗口技术，在许多新型综合软件中得到广泛的应用。本节将向读者提供实现多窗口技术的基本方法和应用多窗口的程序例子。

IBM PC机的ROM例程序，提供了其它一些PC机很少具有的支持多窗口的功能，包括初始化窗口、窗口内容向上卷动和窗口内容向下卷动，它是通过INT 10H的子功能码06和07功能调用而实现的。调用前：

- AH 放子功能码06（上卷）或07（下卷）
- AL 卷动的行数，为0时，实现窗口初始化
- BH 空白区的属性
- CH 窗口左上角的行号
- CL 窗口左上角的列号
- DH 窗口右下角的行号
- DL 窗口右下角的列号

当把窗口左上角的坐标位置放在CX寄存器，右下角的坐标位置放在DX寄存器，空白区的显示属性放在BH之后，用06或07子功能码调用INT 10H，若AL中放0，实现的是清除指定窗口上的内容，即对窗口执行初始化操作。若AL不为0，实现的是把窗口中的内容向上或向下卷动，由AL中指定的行数，向上或向下卷动，将使超出窗口范围的内容消失，从而不会影响相邻窗口中显示的内容，这是窗口独立性的具体体现，窗口底端或顶端新出现的空行，将按指定属性显示空格字符。为了使用上述各功能，可以按如上规

则，用汇编语言实现PASCAL程序的如下一个外部过程：

```
PROCEDURE WINDOW (FUNC,ATTR,UPLEFT,LOWRIGHT:WORD);  
    EXTERN;
```

此处的FUNC代表于功能码，可取值为6或者7。ATTR用于表示窗口的显示属性。UPLEFT和LOWRIGHT分别代表窗口左上角的行列坐标值和右下角的行列坐标值。

在PASCAL程序中，用适当的参数值调用该过程，就能实现初始化窗口和卷动窗口内容的各项操作。

下面给出的是适当简化的该外部过程的汇编语言的源程序清单。PASCAL和用汇编语言实现的外部过程的接口与调用规则、连接方法等，请参阅本书第九章中的有关部分，我们不在这里进行说明。

```
cseg    segment 'CODE'  
assume  cs:cseg  
        public  scroll  
  
scroll  proc far  
        push    bp  
        mov     bp,    sp  
        mov     ah,    6  
        mov     al,    [bp+16]    ; row number to scroll  
        mov     bh,    [bp+14]    ; attribute value  
        mov     dl,    [bp+12]    ; lower right column  
        mov     dh,    [bp+10]    ; lower right row  
        mov     cl,    [bp+ 8]    ; upper left column  
        mov     ch,    [bp+ 6]    ; upper left row  
        int     10h  
  
        mov     ah,    2  
        mov     dh,    [bp+10]    ; lower right row  
        mov     dl,    [bp+ 8]    ; upper left column  
        mov     bh,    0          ; page 0  
        int     10H  
        pop     bp  
        ret     12  
scroll  endp  
cseg    ends  
        end
```

这个外部过程中的各参数值所表示的含义，已用注释形式给出在程序清单中，还可以进一步到下面的PASCAL程序中核实。

要在PASCAL程序中使用多个窗口，首先要定义每一个窗口，即给出每一个窗口所用的显示属性，给出窗口在屏幕上的具体位置(左上角和右下角的坐标值)并进行窗口初始化操作。在使用一个窗口显示某些内容时，常用的方式是，首先把光标放在窗口的左下角

位置，接着开始显示有关内容，并同时计数送往窗口的字符数，当显示字符已到达窗口右端时，执行窗口内容上卷一行的一次操作，并重置光标到窗口左下角位置，继续显示，直到结束。也可以把计数送往窗口的字符数，变为判断光标当前位置是否到达窗口右例，来决定是否要卷动窗口内容。在某些应用场合，也可以使用下卷方案，此时应把光标定位到窗口左上角位置。由于要在用户程序中判断是否到达窗口右侧位置，因此不能在使用多窗口的程序中直接应用PASCAL的标准的READ与WRITE语句，应代之以MS PASCAL语言的系统扩展函数DOSXQQ来完成窗口上的入/出操作。为了在多个窗口上按任意衔接关系完成入/出，还必须在入/出语句中指明所用的窗口。每个窗口定义中还要包括一个本窗口的光标位置信息，在入/出操作过程中，应随时修改光标位置的值。

下面给出一个应用多窗口的程序例子。该程序的功能，是用屏幕上半屏逐行显示一段英文技术资料的原文，用下半屏的左半部分，由运行该程序的人员打入出现在上半屏中的他不认识的英文单词，用下半屏的右半部分给出使用该程序的操作说明。就是说，该程序的功能，是由某人找出一段英文技术资料中他所不认识的单词并显示在屏幕上。为此，要用到三个窗口，第一个窗口用于显示英文资料原文，第二个窗口用于给出使用该程序的操作说明，第三个窗口用于显示用户打入的英文单词。这三个窗口的作用各异，控制入出的手段也不完全相同。对于第一个窗口，让一行英文资料显示在该窗口底部，在显示下一行之前，得首先使窗口内容上卷。程序中得有办法实现在这个窗口上再显示下一行原文的意图，例如，每当用户按一次不属于指定集合的字符键，就实现第一个窗口内容上卷一行，接下来显示英文资料中的下一行原文的操作。这里不能用连续显示完整篇英文资料的方案，而以按用户意图逐行显示为好，否则用户无法看清并打入他不认识的英文单词。为了简化程序中的上卷与显示一行的控制，程序中已规定英文资料每一行的字符数少于80个。对第三个窗口，首先置光标于该窗口的左下角，并接着用于显示输入的属于集合['A'..'Z', 'a'..'z',']的每一个字符，以拼成打入的每一英文单词，同时也可接收空格字符，表示一个英文单词的结束。每向该窗口输出一个字符，要相应修改光标位置值，当光标到了窗口的右侧，需要把该窗口内容上卷一行，并再次定位光标到该窗口的左下角位置。对第二个窗口，把操作该程序的说明信息显示出来之后，在程序运行的期间不再变动。再次强调说明，三个窗口是独立控制的，初始化和上卷任何一个窗口内容，对另外两个窗口内容均无影响，上卷造成的显示内容超出本窗口范围时，超出范围的内容自动消失，不会进入上邻窗口。第三个窗口内容上卷时，也不会影响它的右邻窗口。当用户按功能键F10时，程序结束运行过程，关闭输入文件，并把用户打入过的全部英文单词写到输出文件中。

该程序是为了表明使用多窗口的具体方法和使用效果而给出的，实现的具体功能并无多少实用意义。读者可以看出，就是在这样简单的程序中，离开多窗口技术已经难于简单实现所要求的功能。在更为复杂的应用程序中，多窗口技术可以有更巧妙的用法，产生的效果必将更加引人注目。

下面是该程序的源程序清单。

```
program multyscreen(input,output);

type window=array [1..5] of integer;
    lstr78 =lstring(78);
    validchar= set of char;
```

```

const vchar= ['a'..'z', 'A'..'Z', ' ', ''];
wd1=window (0, 0, 12, 79, 4);
wd2=window(13, 0, 23, 39, 5);
wd3=window (13, 42, 23, 79, 6);
var i,r,c: integer;      t : byte;
    column: string(2);
    fin: text;           lst: string;
    fout: text;          lstr: string(40);
function dosxqq(a,b: word): byte; extern;
procedure scroll(line: integer; wind: window);   extern;

procedure rdwrtn;
begin
    if not eof(fin) then          (* 窗口 1 中的内容上卷一行 *)
        [ scroll(1,wd1);
          readln(fin,lst);        (* 读来输入文件中的一行信息 *)
          write(lst) ]           (* 并显示在窗口1的最底一行上 *)
    end;
begin
    assign(fin,'mltys.pas');      reset(fin);
    assign(fout,'english.wrd');   rewrite(fout);
    write(chr(27), '[J');
    scroll(0,wd1);                (* 初始化窗口1 *)

    scroll(0,wd3);                (* 初始化窗口 3, 并显示操作说明的内容 *)
    write(' window 3');           scroll(2,wd3);
    write(' Operation instructions :'); scroll(1,wd3);
    write(' 1. Enter english word or space'); scroll(1,wd3);
    write('                in window 2'); scroll(1,wd3);
    write(' 2. Enter another char to display'); scroll(1,wd3);
    write(' a line text of fin in window 1'); scroll(1,wd3);

    scroll(0,wd2);    i:=1; t:=0;  (* 初始化窗口 2 *)
    while t< >3 do
        [ repeat t:=dosxqq(6,255) until t<>0;
          if t<>3 then
              if chr(t) in vchar
                  then
                      [ column[1] :=chr(i div 10+48);
                        column[2] :=chr(i mod 10+48);
                        write(chr(27), '[24; ',column,'H')];

```



```

t:=dosxqq(6,t); lstt[i]:=chr(t);
if i mod 39=0
then [ writeln(fout,lstt);
      scroll(1,wd2);
      for i:=1 to 40 do lstt[i]:='',
        i:=1]
else i:=i+1
]
else rdwrtln
];
if i<>1 then writeln(fout,lstr);
close(fin); close(fout)
end.

```

6.5 屏幕页的概念与用法

目前 IBM PC 上配备的显示器有单色显示器和彩色图形显示器两种。多数系统中，单色显示器的显存缓冲区从地址 B0000H 开始，共 4KB。彩色图形显示器的显存缓冲区从地址 B8000H 开始，共 32KB。显示器一屏的显示信息构成一个屏幕页。下面重点介绍彩色图形显示器各屏幕页的用法。

彩色图形显示器有两种基本工作模式：

(1) 字母数字模式 (A/N 模式)

在这种模式下，用两个字节来描述一个字符。一个字节存的是字符的代码值，另一个字节存的是该字符的显示属性。在黑/白显示模式时，字符的属性有加亮、闪烁和反底色显示等。在彩色显示模式下，通常每个字符有 16 种底色和 16 种显示色可以选用，而且仍可用闪烁、加亮等。当每屏上显示 40×25 个字符时，一页占用 $40 \times 25 \times 2 = 2\text{KB}$ 显存。32KB 共可保存 16 个屏幕页的信息，当每屏显示 80×25 个字符时，一页占用 $80 \times 25 \times 2 = 4\text{KB}$ 显存。32KB 共可保存 8 页信息。

(2) 图形显示模式 (APA 方式)

在图形模式下，程序可直接对屏幕上的每个点（又称“象素”或“象元”）进行控制，APA 模式有三种最常用的分辨率：

高分辨率模式： 640×200 点，每点只能取黑白两种颜色。

中分辨率模式： 320×200 点，每点 4 种不同颜色。

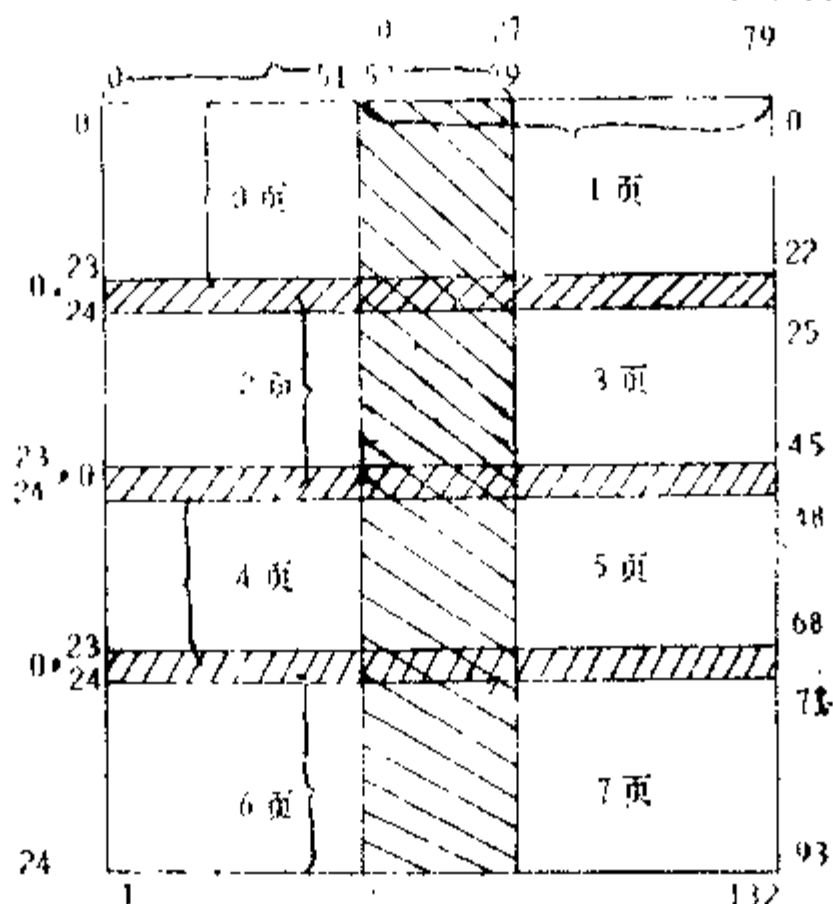
低分辨率模式： 160×100 点，每点 16 种不同颜色。

目前，各种 PC 兼容机之间的三种分辨率模式并不全相同，使用时请查阅有关的手册。

汉字信息显示也是采用图形方式的，但一般终端显示效果并不好，为了获取理想的显示特性，需要采用提高分辨率，扩大显存等方法，这里就不详述了。

对屏幕页的控制，在 PASCAL 程序中可以通过调用相应的汇编语言子程序来实现，PASCAL 程序将显示页的信息作为参数传递给汇编语言子程序，由汇编语言子程序执行相应的软中断调用 (INT 10H)，达到对页面进行控制的功能。关于 10H 中断的详细说明，请参阅有关手册。

下面的例子，是一个简化的西文字符读入的程序，实现了四个方向移动光标、上下左右移动屏幕页、页面重叠处理和读入字符等基本功能。它运用了32KB的显示存储器，八个屏幕页，相邻页面之间采用了部分内容重叠技术，其具体结构如下图所示：



从图上可以看到，每个屏幕页由25行×80字符/行组成。每屏的光标位置，左上角为(0, 0)，右下行为(24, 79)。左右相邻页面之间有28个字符相重叠，即偶数页上每行的第52到79个字符是右邻页上的相应行第0到27个字符，反过来说也一样，即右边一页上每行前28个字符为左邻页相应行中最后28个字符。上下相邻页面之间，上下各有两行重叠，如第2页的前两行与第0页的最后两行重叠，它的最后两行又与第4页的前两行重叠。这样做的目的，是进行移屏操作时，更容易看清和修改屏幕上的内容。

程序中给出了一个94个LSTRING (132) 类型的数组，用于存放读入的字符内容，它和八个屏幕页的对应关系也给在页面结构图中。

程序中用到的全部外部过程，都是用汇编语言实现的。其中：

PAGEN，用于切换屏幕页；

PUTCURS，用于设置光标到指定页的y行x列位置；

GETCURS，用于读指定页的光标所在位置的行列值；

PUTCHATT，把字符按指定的显示属性写到指定页上光标所在位置；

RIGHT，用于把指定页的光标右移一个字符位置；

READKEY，用于读从键盘上打入的一个字符的扫描码和ASCII码。

过程CREENPAGE，用来完成向相邻页面写入字符，是在处理页面重叠部分时要调用的。在向有上邻页的页面中前两行写入时，必须把这些内容写入上邻页的最后两行；在向有下邻页的页面中后两行写入字符时，还必须把这些字符写入下邻页的前两行中。该过程的一个内部过程WRTLTPAGE，用于向左右邻页写入字符，即当向左边一页（编号为偶数）中第52到79列写入字符时，必须把这些字符写入右相邻页同行的前28列的相应位置；同理，当向右边一页（编号为奇数）中第0到27列写入字符时，也必须把这些字符写到左邻页同行中后28列的相应位置。

程序中开始处的for语句，完成定位光标到每一屏幕页的左上角处。p:=0语句用于从

0 页开始读入字符。

程序的主要执行部分，是用一个 REPEAT 语句控制的。开始是读光标位置和读键盘上一个键的扫描码和 ASCII 码。这里的变量 cr 用于给出显示字符时所用的属性，当页编号为 0 时，指定字符颜色值为 7，不为 0 时，指定字符颜色值为页号值。屏幕底色 (Background) 固定为 0 值。

CASE 语句用于按用户所击键的情况执行相应的处理功能，即上下左右移动光标，上下左右移屏和读入字符等功能。在读入字符时，一是要把字符写入 TABLE 数组，二是要把字符显示到当前页面上，三是当读入的字符处在页面重叠部分时，完成向相邻页面的写入操作。移光标用的是 \uparrow \downarrow \leftarrow \rightarrow 四个控制键。移屏用的是 $\overline{\text{F3}}$ $\overline{\text{F4}}$ $\overline{\text{PGUP}}$ $\overline{\text{PGDN}}$ 四个功能键控制的。

程序的最后几行，完成把读入的字符写到 MPAGE.DAT 文件中。

给出该程序的目的，是说明屏幕页的概念和主要用法，而不是给出一个功能齐全的实用的程序。掌握了这里给出的有关知识，在实用程序中使用屏幕页就比较容易了，例如，在实用的屏幕编辑程序中使用了屏幕页，会大大地提高该编辑程序的运行效率。

下面是例子程序的源清单。

```
program table(input,output) ;
var row,col,s,c,p,rw,cl,cr:integer;
    table:array [0..93] of lstring(132) ;
    bell, ch:char;          f: text;

procedure readkey (var s,c:integer) ; extern;
procedure pagen   (page:integer) ;   extern;
procedure putcurs (y,x:integer; page:integer) ;      extern;
procedure getcurs (var y,x:integer; page:integer); extern;
procedure putchatt (page,battr: integer; bchar:char) ; extern;
procedure right   (page:integer) ;   extern;

    procedure screenpage;
var rr: integer;

    procedure wrtirpage(p:integer) ;
begin
    if odd(p)
        then [if col<=27
                then [putcurs(rr, col+52, p-1) ;
                      putchatt(p-1, cr, ch) ; right(p-1) ]
                ]
        else if col>=52
                then [putcurs(rr, col-52, p+1) ;
                      putchatt(p+1, cr, ch) ;      right(p+1) ]
        ]
```

```

end; { wrtlrpage }
begin { screenpage }
  rr:=row; wrtlrpage(n) ;
  if (row<2) and (p>1)
    then [rr:=row+23; putcurs(rr,col,p-2) ;
          patchatt(p-2,cr,ch) ; right(p-2) ;
          wrtlrpage(p-2)
        ] else
  if(row>22) and (p<6)
    then [rr:=row-23; putcurs(rr,col,p+2);
          patchatt(p+2,cr,ch); right(p+2);
          wrtlrpage(p+2)
        ]
  end;
begin { main program }
  bell:=chr(7);
  for p:=0 to 7 do putcurs(0,0,p); n:=0;

repeat
  getcurs(row,col,p);
  if p=0 then cr:=7 else cr:=p;
  readkey(c,c);

case s of
  1: if row>0 then putcurs(row-1,col,p) else write(bell);
  80: if row<21 then putcurs(row+1,col,p) else write(bell);
  75: if col>0 then putcurs(row,col-1,p) else write(bell);
  77: if col<79 then putcurs(row,col+1,p) else write(bell);

  73: if p >= 2 then [p:=p-2; pagen(p)]
        else write(chr(64));
  81: if p <= 5 then [p:=p+2; pagen(p)]
        else write(chr(64));
  61: if not odd(p) then [p:=p+1; pagen(p)]
        else write(chr(64));
  72: if odd(p) then [p:=p-1; pagen(p)]
        else write(chr(64));
  otherwise [cl:=col+1;
             rw:=row-(p div 2)+1;
             if odd(p) then cl:=col-1;

```

```

        if (c>31) and (c<126) and (c1<=132)
            then [ch:=chr(c) ,   table [rw,c1] :=ch,
                  if c1 > ord(table [rw] .len)
                      then table [rw] .len:=wrđ(c1) ,
                  putchatt(p,cr,ch) ,
                  if col<79 then right(p) ,

                  screenpage
                ] else
            if c=13 then [ putcurs(row,0,p) ,
                           if row<24 then putcurs(row+1,0,p) ]
                else write(bell)
            ]
        end,
until s=68,
pagen(0) ,
assign(f,'mpage.dat');   rewrite(f) ,
for row:=0 to 93 do writeln(f,table [row] ) , close(f)
end.

```

下面是用汇编语言实现的外部过程的程序清单。

```

cseg segment 'CODE'
        assume     cs:cseg
        public     putcurs,getcurs,getchatt,putchatt,right
        public     pagen readkey
putcurs proc far
        push       bp
        mov        bp,        sp
        mov        ax,        [bp+10]
        mov        dh,        al
        mov        ax,        [bp+8]
        mov        dl,        al
        mov        ah,        2
        mov        bh,        [bp+6]
        int        10h
        pop        bp
        ret        6
putcurs endp
getcurs proc far
        push       bp
        mov        bp,        sp

```

```

        mov     ah,     3
        mov     bh,     [bp+ 6 ]
        int     10h
        xor     bh,     bh
        mov     bl,     dh
        mov     si,     [bp+10]
        mov     [si] ,  bx
        xor     dh,     dh
        mov     si,     [bp+ 8 ]
        mov     [si] ,  dx
        pop     bp
        ret     6
getcurs endp

pegen   proc     far
        push    bp
        mov     bp,    sp
        mov     al,    [bp+ 6 ]
        mov     ah,    05h
        int     10h
        pop     bp
        ret     2
pegen   endp

right   proc     far
        push    bp
        mov     bp,sp
        mov     bh, [bp+6]
        mov     ah, 3
        int     10h
        inc     dx
        mov     ah, 2
        pop     bp
        int     10h
        ret     2
right   endp

getchatt proc     far
        push    bp
        mov     bp,    sp
        mov     si,    [bp+10]

```

```

        mov     bh,     [si]
        mov     ah,     8
        int     10h
        mov     si,     [bp+8]
        mov     [si],   ah
        mov     si,     [bp+6]
        mov     [si],   al
        pop     bp
        ret     6
getchatt endp

putchatt proc far
        push    bp
        mov     bp,    sp
        mov     bh,    [bp+10]
        mov     bl,    [bp+8]
        mov     al,    [bp+6]
        mov     cx,    1
        mov     ah,    09h
        int     10h
        pop     bp
        ret     6
putchatt endp

readkey proc far
rr:     mov     ah,     1
        int     16h
        jnz     rr
        mov     ah,     0
        int     16h
        push    bp
        mov     bp,    sp
        mov     si,    [bp+8]
        mov     [si],  ah
        mov     si,    [bp+6]
        mov     [si],  al
        pop     bp
        ret     4
readkey endp

cseg    ends
end

```

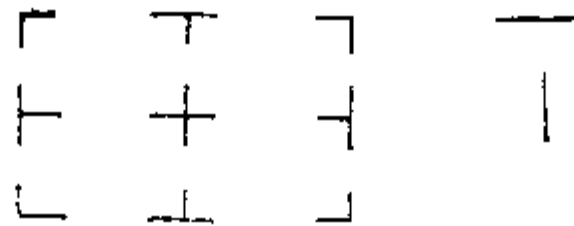
第七章 制表软件和报表支持功能

在进行数据管理的过程中，经常要用到各种各样的表格。这些表格，既可以用来填入有关数据形成各类报表，也可以用来在屏幕上配合显示录入的数据。怎样能更方便地绘制和管理这些表格，怎样更合理地使用这些表格，是本章要讨论的几个问题，重点放在介绍不同的制表方案方面。最后还要讨论屏幕上绘制图形的基本技术。

7.1 制表软件实现的基础

所谓制表，就是用PC机终端提供的11个制表符，绘制出所用的各种表格的过程。除此之外，还要在空表上填上表格的标题，各标题栏的标题栏目，确定表中将要填入的数据的类型和准确位置等等。

11个制表符是：



这11个制表符，在IBM/PC、长城0520微机的西文DOS系统中，可以用小键盘上的11个键的上档键经变换后打入，它们的ASCII值都大于128，即每个制表符用一个字节表示，该字节的最高一位的值是1。而在WANG/IPC微机，或在IBM/PC、长城0520微机的中文DOS系统中，这些制表符与汉字符号一样，在计算机内是用两个字节表示的，在显示或打印时，要占用两个字符的宽度。

制表符是用一个字节表示（显示时只占一个字符宽度），还是用两个字节表示（显示时占两个字符宽度），对形成和使用表格是有较大影响的。对后一种情况，必须特别当心出现半个制表符的情形，处理不当，会成为相当烦人的一项错误。

形成一张表格最基本的问题，是安排和处理11个制表符之间的关系。请看下表：



这是一张故意把每个制表符与相邻的制表符分开一点画出来的表格例子。从图中可以看到：

• 四个小拐角形制表符，只出现在一张大表的四个角上（或表内可能出现的相应拐角处）；

• 四个T字形制表符，只出现在一张大表的外框与表内横线、竖线相交处（或表内的相同局面之处）；

• 十字形制表符，只出现在表内横竖线交叉的位置；

• 制表符一和|，分别用于绘制横线和竖线中那些不与其它制表符相接的部分。

实际上，制表程序正是按上述原则来确定一张表格中每一个位置上的制表符的，无制表符的位置用空格字符。

下面给出的第一个程序，是在IBM/PC、长城0520微机的西文DOS系统下，用来绘制一张简单的表格程序例子。第二个程序，是在WANG/IPC机上实现类似功能、支持汉字入/出的程序例子。

第一个程序

```
program sedtl (input,output) ;
const
  line=24;
  width=40;
type
  arrc=packed array [1..line,1..width*2] of char;
var
  ar:arrc;
  esc:char;
  f,fd:text;
  i, r, c, top,left, bottom,right,dish,discv:integer;

  procedure datoch (i:integer) ;
    var ch:char;
  begin
    ch:=chr (48+i div 10) ; write (fd,ch) ;
    ch:=chr (48+i mod 10) ; write (fd, ch) ;
    writeln (fd)
  end;

begin
  (*set initial value of variables*)
  esc:=chr (27) ;
  for r:=1 to 24 do
    for c:=1 to 80 do ar [r,c] :='';
  writeln (esc,' [2J') ;
  write (esc,' [ 2; 12H', esc, ' [32;44m',
```

TABLE-MAKING SOFTWARE', esc, ' [0m') ;

(* provide parameters *)

```
write (esc, ' [4;12H', 'input parameters:')
write (esc, ' [5;8H', 'up-left line number:') ; readln (top) ;
write (esc, ' [6;8H', 'up-left column number:') ; readln (left) ;
write (esc, ' [7;8H', 'down-left row number:') ; readln (bottom) ;
write (esc, ' [8;8H', 'down-right column number:') ; readln (right) ;
write (esc, ' [9;8H', 'distance of horizon:') ; readln (dish);
write (esc, ' [10;8H', 'distance of vertical:') ; readln(disv);
```

(* create a table *)

for r:=top to bottom do

for c:=left to right do

if r=top

```
then if c=left then ar [r,c] := '┌' else
      if c=right then ar [r,c] := '┐' else
      if (c-left) mod (disv+1) = 0
      then ar [r,c] := '┴'
      else ar [r, c] := '—'
```

else

if r=bottom

```
then if c=left then ar [r,c] := '└' else
      if c=right then ar [r,c] := '┘' else
      if (c-left) mod (disv+1) = 0
      then ar [r,c] := '┴'
      else ar [r,c] := '—'
```

else

if (r-top) mod (dish+1) = 0

```
then if c=left then ar [r,c] := '┌─' else
      if c=right then ar [r,c] := '─┐' else
      if (c-left) mod (disv+1) = 0
      then ar [r,c] := '┴'
      else ar [r,c] := '—'
```

else

if (c=left) or (c=right) or ((c-left) mod (disv+1) = 0)

```
    then ar [r,c] := ' | ';
```

```
    (* displsy the table on screen *)
```

```
write (esc, ' [ 2 J' ) ,
```

```
for r:= 1 to bottom do
```

```
    [if r>=top then
```

```
        for c:= 1 to right do write (ar [r,c] ) ,
```

```
        writeln
```

```
    ] ,
```

```
    (* write the contents of table into file *)
```

```
writeln ('wait ! write the contents of table into file' ) ,
```

```
assign (f, 'format.tbl' ) ; rewrite (f) ,
```

```
for r:=1 to bottom do
```

```
    [ for c:=1 to right do write (f,ar [r,c] ) ,
```

```
    writeln (f) ] ,
```

```
close (f) ,
```

```
assign (fd, 'format.fmt' ) ; rewrite (fd) ;
```

```
    writeln(fd,top,bottom),
```

```
    writeln (fd,left,right) ,
```

```
    writeln (fd, dish,disc) ;writeln (fd) ,
```

```
close (fd)
```

```
end.
```

第二个程序

```
program sedt (input, output) ,
```

```
const
```

```
    line=24,
```

```
    width=40,
```

```
type
```

```
    arrc=packed array [ 1..line,1..width * 2 ] of char,
```

```
    arr= array [ 1..line,1..width ] of string ( 2 ) ,
```

```
var
```

```
    ld:adr of arr ; ! 给出两个地址类型ld和cd是为了用不同
```

```
    cd:adr of arrc ; ! 的方法使用同一块内存区, 即arr变量区
```

```
    ar:arr ;
```

```
    esc:char ;
```

```

f:text;
r,c,top, left,bottom,right,dish,disv: integer;

begin
    (* 设置变量初值 *)
    ld:=adr ar; cd.r:=ld.r;
    for r:= 1 to 24 do
        for c:= 1 to 80 do cd^[r,c]:= '';

    esc:=chr (27) ;
    write[ chr (12) , esc, '[ 8;12H', esc, '[ 1, 4m') ;
    write ('制表软件', esc, '[0m');

    (* 提出制表要求 *)
    write (esc, '[10, 8H', '输入制表要求:');
    write (esc, '[12, 8H', '左上角行号:'); readln (top) ;
    write (esc, '[13, 8H', '左上角列号:'); readln (left) ;

    write (esc, '[15, 8H', '左下角行号:'); readln (bottom) ;
    write (esc, '[16, 8H', '左下角列号:'); readln (right) ;

    write (esc, '[18, 8H', '横线距离:'); readln (dish) ;
    write (esc, '[19, 8H', '竖线距离:'); readln (disv) ;

    (* 生成表格的内容 *)
    for r:=top to bottom do
        for c:=left to right do
            if r=top
                ! 外框上边线
                then if c=left then ar [r,c] :='┌' else
                    if c=right then ar [r,c] :='┐' else
                        if (c-left) mod disv=0
                            then ar [r,c] :='┴'
                            else ar [r,c] :='—' else

            if r=bottom
                ! 外框下边线
                then if c=left then ar [r,c] :='└' else
                    if c=right then ar [r,c] :='┘' else
                        if (c-left) mod disv=0
                            then ar [r,c] :='┴'
                            else ar [r,c] :='—' else

```

```

if (r-top) mod dish=0          ! 框内各横线
then if c=left then ar[r,c] := '┌' else
    if c=right then ar[r,c] := '└' else
    if (c-left) mod disv=0
        then ar[r,c] := '+'
        else ar[r,c] := '-' else

if (c=left) or (c=right) or ((c-left) mod disv=0)
then ar[r,c] := '|';          ! 框内各竖线

```

(* 表格内容写进文件 *)

```

assign (f, 'format.tbl'); rewrite (f);
for r:=1 to bottom do
    [for c:=1 to right*2 do write (f, cd^ [r,c] );
     writeln (f) ];
close (f);

```

(* 表格内容显示到终端屏幕上 *)

```

write (chr(12));
for r:=top to bottom do if r<=24 then
    [write (esc, ' [', r:2, '; ', left*2-1:2, 'H');
     for c:=left*2-1 to right*2 do
         if c<=80 then write (cd^ [r,c] )
    ]

```

end.

上两个程序很类似，许多语句是相同的。但更应该看到二者的不同之处，主要表现在：

(1) 用的一个数组，都是 24×80 个字符的大小。在第一个程序中，就用 $1..24$ 和 $1..80$ 作为下标来说明数组，每个制表符在那里被作为单个字符使用，而在第二个程序中，这个数组是用 $1..24$ 和 $1..40$ 作为下标，并被说明是基类型为STRING(2)的数组。在那里，每个制表符要用两个字节表示，就用STRING(2)的类型来表示它们，这可以防止出现半个制表符的错误。在输入制表参数值时，涉及到列数值的地方，用的都是汉字的列数。如表的左部准备空10个字符位置，左上角列号应为6，又如竖线距离为2，表示的是两条竖线间能写得下两个汉字或四个英文字母。

(2) 在制表过程中，或者在使用表格的过程中，表格内可以填入汉字，也可能填入数字或英文字母等。此时，对第二个程序，用STRING(2)作为基类型来使用数组就不方便了。为此，在第二个程序中，通过给出两个地址类型LD和CD，实现用两种不同的方法来使用同一片内存区的目的。处理制表符时，直接在AR数组上操作。在需要处理单个字符时，用 $cd^{\uparrow} [r,c]$ 的方式操作。对第一个程序，不存在使用汉字问题，无此

矛盾，处理起来略简单一点。

这两个程序的核心部分是生成表格的全部内容。采用的算法可简述如下：

- 在设置变量初值时，把ar变量所用的一片内存区全清成空格字符。

- 用给出的表格外框的行号（从TOP到BOTTOM）和列号（从LEFT到RIGHT）

执行二重循环，把有关的制表符写到ar变量的相应单元中，具体规则，就是我们在本节开始讲解11个制表符的相互关系时给出的规则。在程序中，是逐行确定表格每个位置上制表符的。我们把各行分成四类：

最顶上一行，

最底下一行，

有横线的各行，

无横线的各行。

对每一行，我们又把它分成左边框、右边框、对应有竖线的各列和无竖线的各列，并按照行列情况确定每个位置上的具体制表符。讲清这些，有助于每位读者看懂语句虽然不多，但功能却比较丰富的这一段程序（其实只写了一个大的复合语句）。

制好的空表内容，还要显示在屏幕上，写到磁盘上指定的文件中。第一个程序，还把表参数的六个值写到磁盘上另一个文件中。这些语句比较容易看懂，不再解释。

在我们的程序中，表格大小正好等于终端屏幕一屏所能显示的内容，其实，表格完全可以按用户需要定义，行数可以更多，列数也可以更多。对我们上面的程序，只要修改数组大小并修改控制显示表格内容的几个语句，加上上下滚动屏幕内容、左右移动显示窗口的部分就可以了。实用表格的大小，主要受所用打印机的限制。

7.2 制表软件的进一步完善

上节给出的程序，仅完成绘制一张空表的功能，要达到实用的目标，还必须增加修改已有表格的功能（即补一些横、竖线，或去掉某些横、竖线）、增加填写表格标题、标题栏名目的功能，以及规定向表格中写入的每一个数据的类型、写入的准确位置及写入的格式等功能。下面给出的程序就实现了这些功能，它要在上节第一个程序运行之后，已经建立了FORMAT.TBL文件和FORMAT.FMT文件的基础才能运行，它要读进这两个文件的内容作为原始数据。它是在IBM/PC西文方式下运行的一个例子。

```
program fillin (input, output),
const
    line=24, width=80;

type
    arr=array [1..line, 1..width] of char;
    lst=lstring (20);

var
    ar:arr          ; ftab, ffmt:text;
    r,c:integer     ; esc, bell,ch:char;
    w:word          ; scan,asc:byte;
    top,bottom, left,right,diso,dis:integer;
```



```

procedure drawhline;
begin
    position (21, 11) ; write (esc, ' [K') ;
    write ('line number, start, ends: ') ;
    readln (r, start, ends) ;
    position (r, start) ;
    for c:=start to ends do
        [if c=start then changech (1) else
         if c=ends then changech (3)
         else changech (2) ;
         write (ar [r,c] ) ]
end;

```

```

procedure drawvline;
begin
    position (21,11) ; write (esc, ' [K') ;
    write ('column number, start, ends: ') ;
    readln (c, start, ends) ;
    for r:=start to ends do
        [if r=start then changech (7) else
         if r=ends then changech (9)
         else changech (8) ;
         position (r, c) ; write (ar [r, c] ) ]
end;

```

```

procedure eraseh;
begin
    position (21, 11) ; write (esc, ' [K') ;
    write ('line no., start, ends: ') ;
    readln (r'start, ends) ;
    position (r, start) ;
    for c:=start to ends do
        [if c=start then changech (4) else
         if c=ends then changech (6)
         else changech (5) ;
         write (ar [r, c] ) ]
end;

```

```

procedure erasev;

```



```

begin
    position (21, 11) ; write (esc, ' [K') ;
    write ('column no., start, ends: ') ;
    readln (c, start, ends) ;
    for r:=start to ends do
        [if r=start then changech (10) else
         if r=ends then changech (12)
          else changech (11) ;
         position (r, c) ; write (ar [r, c] ) ]
end;

procedure sedt (d:integer) ;
begin
    repeat
        bo:=false;
        cursorpos (r, c) ;
        readcode;
        if (scan in [wrd (71) ..81] ) and (asc in [wrd (43)..57] )
            then [ch:=graphtab [scan-70] ;
                 ar [r,c] :=ch; write (ch) ] else
            if asc=0 then
                [ if scan=72 then write (esc, ' [A') else
                  if scan=80 then write (esc, ' [B') else
                  if scan=75 then write (esc, ' [D') else
                  if scan=77 then write (esc, ' [C') else
                  if (scan>=59) and (scan<=66)
                      then bo:=true
                      else write (bell)
                ] else
                if ( (d=0) and not (chr (asc) in imtset) )
                  or ( (d=1) and (chr (asc) in fmtset) )
                  then [at [r, c] :=chr (asc) ; write (chr (asc) ) ]
                  else write (bell)
            until bo;
    end;

    procedure prompt;
    var
        ch:char;
        i, j:integer;
    begin

```

```

write (esc, ' [22; 05H', ' F1:draw horizontal line',
      esc, ' [22;35H', ' F3:draw vertical line',
      esc, ' [23;05H', ' F2:erace horizontal line',
      esc, ' [23;35H', ' F4:erace vertical line',
      esc, ' [24;05H', ' F5:giving table title',
      esc, ' [24;35H', ' F6:define data format',
      esc, ' [25;05H', ' F7:save resultat in file',
      esc, ' [25;35H', ' F8:finish this screen operation')
end,

```

```

procedure prt (lstmsg:lst) ;
begin
  write (esc, ' [24;64H', esc, ' [31;42m',
        lstmsg, esc, ' [ 0 m')
end,

```

```

procedure writefile;
  var startend:boolean;
  typ, line, lenth:integer;
begin
  rewrite (ftab) ; rewrite (ffmt) ;
  writeln (ffmt, top:4, bottom:4, left:4, right: 4) ;
  startend:=false;
  for r:=1 to bottom do
    [for c:= 1 to right do
      [ch:=ar [r, c] ,
        if not (ch in fmtset)
        then write (ftab, ch)
        else [write (ftab, ' ' ) ,
              startend:=not startend;
              if startend then
                [case ch of
                  '! ':typ:= 1,
                  '^':typ:= 2,
                  '? ':typ:= 3
                end,
                line:=r, start:=c
              ] else
                [lenth:=c-start+ 1,
                  writeln (ffmt, typ:4, line: 4,

```

```

start: 4, length: 4) ]
    ]
  ], writeln (ftab)
],
close (ftab) ; close (ffmt)
end;

```

```

begin (*MAIN PROGRAM*)

```

```

  esc:=chr (27) ; bell:=chr (7); bo:=false;
  for r:= 1 to line do
    for c:= 1 to width do ar [r, c] := ' ',

```

```

  (* repeat
    readcode,
    writeln(scan,asc)
    until asc= 3, *)

```

```

  (*FIND PARAMETERS*)
  assign(ffmt, 'format.fmt'); reset(ffmt);
  readln(ffmt, top, bottom);
  readln(ffmt, left, right);
  readln(ffmt, diso, disv);
  close(ffmt);

```

```

  (*READ TEXTFILE INTO ARRAY*)
  assign(ftab, 'format.tbl'); reset(ftab);
  r:= 0;
  while not eof(ftab) do
    [r:=r+ 1; c:= 0;
    while not eoln(ftab)do
      [c:=c+ 1; read(ftab, ar [r, c] )];
    if not eof(ftab)then readln(ftab)
    ];
  close ftab);

```

```

  (*DISPLAY THE TABLE*)
  write(esc, ' [2J') ;
  for r:= 1 to bottom do
    [for c:= 1 to right
      do write(ar [r, c] );

```

```

        writeln] ;

prompt;
repeat
    prt('root screen') ;
    if scan=66 then bo:=not bo;
    if bo then bo:=not bo
        else readcode;

    if (asc=0) and (scan>=59) and (scan<=66) then
        case scan of
            59: [prt('drawhline'), drawhline] ;
            60: [prt('eraseh'), eraseh] ;
            61: [prt('drawvline'), drawvline] ;
            62: [prt('erasev'), erasev] ;
            63: [prt('def title'), sedt(0)] ;
            64: [prt('data form'), sedt(1)] ;
            65: [prt('write file'), writefile] ;
            66: ;
            otherwise
                end else write(bell);
        until (asc=3) or ((scan=66) and not bo);
    end.

```

这个程序的功能多一些，比较长，但不是太难读懂。我们准备从三个方面进行说明。

第一，修改表格的操作，是由补加横线、去掉横线、补加竖线、去掉竖线的四个过程 DRAWHLINE、ERASEH、DRAWVLINE、ERASEV 分别完成的。主程序部分接收到 F1、F2、F3 或 F4 键输入时，将引起调用这四个过程。进入过程后，要求用户指定行号或列号，以及一条线的列或行的起始位置 START 和结束位置 ENDS，然后按此要求，把相应的制表符写入 ar 数组的对应单元，并显示在屏幕对应的位置。这里的关键问题，是找出在表的同一位置上，过程运行前已有的制表符和过程运行后应得到的制表符之间的对应关系。这种对应关系是比较复杂的，必须区分运行的是哪一个过程，又必须区分在补加、去掉一条线的过程中，当时处理的是什么位置（线的起始位置、中间位置或结尾位置）。我们按这 12 种不同的情况，把各种制表符的对应关系填在数组 CHANGE 中。该数组由 13 行 STRING (12) 类型的数据组成。change [0] 中给出的是过程运行前，表中某一位置可能取的制表符（共十一种），或是一个非制表符（此处用空格代表，实际上也可以是别的字符）。以下的 12 行数据，是 12 种情形下，运行过程后应得到的制表符，它是运行过程、线的位置、运行前相应位置上原有的制表符三项内容的一个函数。四个过程各用三行数据，分别表示线的起始、中间、结尾三种情形。在查询该数组，以完成字符变换的过程中，运行过程和线的三种位置是通过不同的数组下标体现出来的，其值由主程序给出。真正的变换是通过调用过程 CHANGECH 完成的，它检查表格一确定位置上原有制

制表符与CHANGE [0] 的哪一个位置的制表符相同，求得STRING (12) 的一个下标值（不为制表符时，该下标值一定为12），然后把该数组中指定下标处的数据的对应位置上的制表符，写在表示表格的数组中。举个例：假定要运行的过程是 DRAWHLIN (补加一条横线)，该线起始位置、中间若干位置、结尾位置的制表符均为|，则应查CHANGE 的1号、2号和3号三个单元，查到的数据的位置值都为11，则得到的制表符将分别为|、+和-，对那些有非制表符的位置，查得的位置值均为12，得到的制表符都应是一。其效果可表示如下：

过程运行前相应行为：

过程运行后该行变为：

—— — — — — — — — — —

（这条横线实际上是连起来的，画成这种样子是为了看得清楚些）。

第二，确定表格标题和将来要登入数据的类型、位置和格式的操作，是通过类似于屏幕编辑的一小段程序完成的。在这一操作过程中，既能填表格标题，也能填确定数据格式的信息，还能修改表格。要解决的问题是，必须区分表格标题内容和数据格式内容，这是通过对两种内容使用不同的字符集合来实现的。例如，规定标题中不准使用'?' '1' '@' 等字符，就可以用这些字符来代表数据类型，并用它们在表格中的位置表示写入数据的位置。表示数据格式，主要是规定小数点在实数中的位置，以及较长字符串的自动折行等，为了简便，我们在程序中省略了这后一部分内容。整个具体方案可以通过读程序了解清楚。


第三，结果写入文件的操作，要由两部分内容构成，一是把表格及标题写入表格文件，在写的过程中，把定义数据格式的字符都换成空格写入该文件。二是把定义数据格式的字符转换成数字写入数据格式文件。每一格式信息将由数据类型、起始位置（行、列值）、数据长度三项数据组成。这是控制向表格填入数据的依据。

请注意，这个程序是实现真正实用的制表软件的骨架，带有明显的入门性质，目的在于让读者了解设计制表软件的方案和可用技术，而没有涉及实用中必须解决的诸多细节问题。即使如此，作为讲解软件实现的原理的例子，这个程序已经很长了。

7.3 屏幕编辑方式制表

前两节给出的制表方案，都是用 READ 语句输入某些参数值，再按给出的值得出相应的表格内容，有点类似于编辑程序中的行编辑或称命令方式的编辑程序。本节将给出另外一种制表方案，它是以全屏幕编辑方式实现制表的，也就是说，通过上下左右移光标，在光标经过的位置画出相应的制表符，每一条线都是光标移动的轨迹。这种办法特别适用于绘制那些非常不规则的表格。

这种制表方式的实现原理，就是记下光标移动的轨迹（记录在表示表格的数组中，并显示到屏幕上），容易理解，但是要达到方便实用的目标，其判断条件是相当复杂的，请看下述的几种情况：

光标轨迹	制表内容	说 明
(1) 		开始时表格内容为空，只在光标移动方向发生变化时，得到不同的拐角形制表符，否则只能是一或 。

如果光标移动方向发生变化，就用小拐角形制表符，则在①处得到：

①

如果光标移动方向不变，就用—或|，则在②、③和④处分别得到，

正确的结果应为：①处应由一变为上，②处应由┘变为上，③处应由！变成+，④处应由一变成+。

在说明了上述问题之后，我们给出一个简单的原理性的制表程序，它离实用目标相差较远，但用于表明实现制表的原理是足够清楚的。下面是该程序的源清单。

```

program drawtab (input, output) ,
const
    line=24,      width=80,
    graphtab='  □ ▢ ▣ ▤ ▥ ▦ ▧ ▨ ▩ □ ▢ ▣ ▤ ▥ ▦ ▧ ▨ ▩ □ ▢ ▣ ▤ ▥ ▦ ▧ ▨ ▩',
type
    arr=array [1..line, 1..width] of char;

```

var

ar:arr; f, c, i:integer;
w:word; esc, bell, ch:char;
 scan, asc:byte;

function keyin:word; extern;

procedure cursorpos (var y,x:integer) ; extern;

procedure readcode;

begin

 w:=keyin;

 scan:=hibyte (w) ; asc:=lobyte (w)

end;

procedure position (y,x:integer) ;

begin

 write (esc, ' [' ,y div 10:1,y mod 10:1, ',' ,
 x div 10:1,x mod 10:1, 'H')

end;

procedure drawtable;

 var draw:boolean;

 odirect,pdirect: (no,up,down,left,right) ;

begin

 draw:=false; odirect:=no;

 repeat

 cursorpos (r,c) ; (* 测定光标位置 *)

 readcode; (* 读来输入键的扫描码和ASCII码 *)

 if (scan in [wrd (71) ..81]) and (asc in [wrd (43) ..57])

 then [ch:=graphtap [scan-70] ;

 ar [r,c] :=ch; write (ch)] else (* 制表符 *)

 asc=0 then

 [if scan=72 then

 [if draw then (* 移光标 *)

 [pdirect:=up;

 if odirect=left then ar [r,c] :='[' else

 if odirect=right then ar [r,c] :=']' else

 if ar [r,c] ='-' then ar [r,c] :='+'

```

                                else ar [r,c] := ' | ' ;
                                write (ar [r,c] ,esc, ' [D') ] ;
                                write (esc, ' [A') ] else
if scan=80 then
    [if draw then
        [pdirect:=down,
            if odirect=left then ar [r,c] := ' [ ' else
            if odirect=right then ar [r,c] := ' ] ' else
            if ar [r,c] = ' — ' then ar [r,c] := ' + '
                                else ar [r,c] := ' | ' ;
            write (ar [r,c] ,esc, ' [D') ] ;
            write (esc, ' [B') ] else

if scan=75 then
    [if draw then
        [pdirect:=left,
            if odirect=up then ar [r,c] := ' . ] ' else
            if odirect=down then ar [r,c] := ' ] ' else
            if ar [r,c] = ' | ' then ar [r,c] := ' + '
                                else ar [r,c] := ' — ' ;
            write (ar [r,c] ,esc, ' [D') ] ;
            write (esc, ' [D') ] else

if scan=77 then
    [if draw then
        [pdirect:=right,
            if odirect=up then ar [r,c] := ' [ ' else
            if odirect=down then ar [r,c] := ' [ ' else
            if ar [r,c] = ' | ' then ar [r,c] := ' + '
                                else ar [r,c] := ' — ' ;
            write (ar [r,c] ,esc, ' [D') ] ;
            write (esc, ' [C') ] else

if (scan>=59) and (scan<=68) then draw:=not draw,
if not draw then pdirect:=no, (* 功能键 *)
odirect:=pdirect
] else write (bell)
until scan=68, (* F10键结束该过程 *)
end,

```



```

begin (*MAIN PROGRAM*)
  esc:=chr (27); bell:=chr (7);
  for r:=1 to 24 do
    for c:=1 to 80 do ar [r,c] := ' ',
  write (esc, ' [J');
  drawtable
end.

```

在这个程序中，过程 drawtable 是完成屏幕制表的。局部变量 draw 是用于表明光标移动时要不要画线的一个布尔型变量，它为真，表示要画线，为假，不要画线。它的取值通过按F1到F10这些功能键来改变。这种控制是必不可少的，否则在整个画线过程中，就无法实现只移光标不画线的操作。成为如同手工画线时不得“抬笔”的困难局面。odirect和pdirect是用于表示前一次光标移动方向和本次移动方向的枚举类型的变量，取值除上下左右之外，还有一个no，表示无方向。为了记录不存在前一次光标移动方向的情形，要使odirect取值为no。本程序刚开始运行时，或不画线只移光标操作刚结束之后都属于这种情形。

在这个过程内，首先是测定光标位置，接着读来输入键的扫描码和字符编码。按输入键的情况，分成三种处理操作：

- (1) 当输入的键为制表符时，则接收输入并同时把该制表符显示在光标所在位置；
- (2) 当输入的键为↑↓←→时，则应完成：
 - 记下光标移动方向；
 - 比较本次和前一次光标移动方向，需要时，给出相应的小拐角形制表符，否则检查光标位置的已有制表符，以决定应给出的制表符。在该程序中，左右移光标时遇竖线，或上下移光标时遇横线，新制表符均为+，是一种简化的处理；
 - 输出新的制表符后再使光标位置复原；
 - 按要求移动光标；
 - 最后还要把本次光标移动的方向赋给odirect，作为下次移光标时的前一次光标移动方向。

(3) 当输入的为F1~F10这些功能键时，要使draw取它的反值。当draw为FALSE时，要使pdirect为no，下一个语句将使odirect也取值no。

整个过程是用REPEAT语句控制的，当输入F10键时，过程结束。

程序本身只有向变量赋初值，清屏和调用drawtable几个语句，不必解释。对整个程序的功能，还必须进一步解释如下：

首先，为了使程序尽可能的简明，对制表符的检查和判断是很不充分的，在制表过程中，存在本节开始指出的一些问题，可能出现线上的小缺口，也可能出现不正确的制表符。此时可以进入只移光标不画线的状态，用把光标移到有错的位置后键入正确的制表符的办法改正错误，这是最简便的解决措施。

第二，在应用中，不仅要求能用记下光标轨迹的办法画线，也要求能用这种办法擦线，这是修改错画的线所必须的。我们的程序中没有给出这种功能，实现的原理和画线类似，我们相信读者有能力自行实现。

第三，用这种方式制表，同样存在给出表格标题，定义数据类型、位置和格式等操作


```

function keyin:word, extern;
procedure cursorpos (var y,x:integer) ; extern;

procedure readcode;
begin
  w:=keyin;
  scan:=hibyte (w) ; asc:=lobyte (w)
end;

procedure position (y,x:integer) ;
begin
  write (esc, ' [', y div 10:1, y mod 10:1, '; ',
        x div 10:1, x mod 10:1, 'H')
end;

procedure change;
begin
  if draw then
    [if (not odd (scan) and (ar [r,c] = '—') ) or
      (odd (scan) and (ar [r,c] = ' | '))
      then ar [r,c] := '+'
      else ar [r,c] := table [ord (pdirect)] [ord(odirect)+ 1] ,
      write (ar [r,c] ,esc, ' [D')
    ] ;
  write (esc, ' [', table [ord (pdirect) ] [6] )
end;

procedure drawtable;
begin
  draw:=false; odirect:=no; pdirect:=no;
  repeat
    cursorpos (r,c) ;
    readcode;
    if (scan in[wrđ (71) ..81] ) and (asc in[wrđ (43) ..57])
      then[ch:=graphtab[scan-70]; ar[r,c]:=ch; write (ch) ]
      else
        if asc= 0 then
          [if scan=72 then [pdirect:=up ; change] else
            if scan=80 then [pdirect:=down; change] else
              if scan=77 then [pdirect:=right; change] else

```

```

        if scan=75 then [pdirect:=left; change] else
        if (scan>=59) and (scan<=68) then draw:=not draw;
        if not draw then pdirect:=no;
        odirect:=pdirect
    ] else write (bell)
until scan=68;
end;

begin (* MAIN PROGRAM *)
    esc:=chr (27) ; bell:=chr (7) ;
    for r:=1 to 24 do
        for c:=1 to 80 do ar[r,c]:='';
    write (esc,'[J') ;
    drawtable
end.

```

7.4 程序中直接形成表格

前两节都是用制表软件形成表格内容，并且在磁盘上建立表格文件。当另外的程序要用某一表格时，首先要从磁盘上读来表格内容和数据格式的内容后才能使用。对有些简单的表格，也可以在程序中直接形成。这实际上是把本章第一、二两节的内容合到一起，做成一个支持制表的UNIT，供其它程序调用。例如，它提供如下六个关键过程。

(1) 制一张规则的空表的过程，用到的参数是表格外框左上角、右下角的行、列值，表内横、竖线的距离，它是本章第一节给出的程序中的部分内容。

(2) 到(5)分别为补加一条横线，去掉一条横线，补加一条竖线，去掉一竖条线的四个过程，用到的参数是一个行或列值，以及在行列上起始的和终止的列或行号，这是本章第二节的程序中的部分内容。

(6) 给出表格标题或标题栏名目的过程，用到的参数是标题所在的行和起始列号，以及标题内容本身。

从程序清单上可以看到，前五个过程，只是简单地把前面给出的程序中用读语句读来的几个值，换成过程参数得到的。给出标题的过程，是按给出的行列值，把标题内容写进数组相应位置并同时显示在屏幕上。

在程序中给出的用于存放表格内容的数组，六个过程都要用到，必须把它作为每个过程的变量参数。

下面给出该UNIT的实现部分和接口部分，最后是使用UNIT的一个程序例子。程序只是形成和显示表格，无其它功能，运行结果容易看清楚，不在这里给出。

```

(* $include, 'flmodu.inf' *)
implementation of flmodu;

```



```

for c:=left to right do
  if r=top
    then if c=left then ar [r,c] := '┌' else
          if c=right then ar [r,c] := '┐' else
          if (c-left) mod (disv+1) = 0
            then ar [r,c] := '└'
            else ar [r,c] := '─'
    else
  if r=bottom
    then if c=left then ar [r,c] := '└' else
          if c=right then ar [r,c] := '┐' else
          if (c-left) mod (disv+1) = 0
            then ar [r,c] := '┌'
            else ar [r,c] := '─'
    else
  if (r-top) mod (dish+1) = 0
    then if c=left then ar [r,c] := '┌─' else
          if c=right then ar [r,c] := '─┐' else
          if (c-left) mod (disv+1) = 0
            then ar [r,c] := '+'
            else ar [r,c] := '─'
    else
  if (c=left) or (c=right) or ((c-left) mod (disv+1) = 0)
    then ar [r,c] := '|';
write (esc, ' [2J') ;
for r:=1 to bottom do
  [for c:=1 to right
    do write (ar [r,c]) ;
  write[n]
end;

procedure drawhline;
begin
  position (r,start) ;
  for c:=start to ends do
    [ch:=ar [r,c] ;
    if c=start then ar [r,c] :=changech (1,ch) else
    if c=ends then ar [r,c] :=changech (3,ch)
    else ar [r,c] :=changech (2,ch) ;
    write (ar [r,c]) ]

```

end;

procedure drawvline;

begin

for r:=start to ends do

[ch:=ar[r,c];

if r=start then ar[r,c]:=changech(7,ch) else

if r=ends then ar[r,c]:=changech(9,ch)

else ar[r,c]:=changech(8,ch);

position(r,c); write(ar[r,c])]

end;

procedure eraseh;

begin

position(r,start);

for c:=start to ends do

[ch:=ar[r,c];

if c=start then ar[r,c]:=changech(4,ch) else

if c=ends then ar[r,c]:=changech(6,ch)

else ar[r,c]:=changech(5,ch);

write(ar[r,c])]

end;

procedure erasev;

begin

for r:=start to ends do

[ch:=ar[r,c];

if r=start then ar[r,c]:=changech(10,ch) else

if r=ends then ar[r,c]:=changech(12,ch)

else ar[r,c]:=changech(11,ch);

position(r,c); write(ar[r,c])]

end;

procedure title;

var cl:integer;

begin

position(r,c);

for cl:=1 to ord(st[0]) do

[ar[r,c+cl-1]:=st[cl]; write(ar[r,c+cl-1])]

```

    end,
end.

INTERFACE,
unit flmodu (line,width,arr,lst,frame,drawhline,
            drawvline,eraseh,erasev,title) ,

const
    line=24,                width=80,

type
    arr=packed array [1..line,1..width] of char,
    lst=lstring (80) ,

    procedure frame      (top,left,bottom,right,dish,dishv:integer,
                        var ar:arr) ,
    procedure drawhline (r,start,ends: integer,    var ar:arr) ,
    procedure drawvline (c,start,ends: integer,    var ar:arr) ,
    procedure eraseh     (r,start,ends: integer,    var ar:arr) ,
    procedure erasev     (c,start,ends: integer,    var ar:arr) ,
    procedure title     (r,c :integer, st: lst,    var ar:arr) ,
end,

    (* $include:'flmodu.inf' *)
program asdf (input,output) ;
    uses flmodu (line,width,arr,lst,frame,drawhline,
                drawvline,eraseh,erasev,title) ;
var
    ar:arr,
    r,c:integer,

begin
    for r:=1 to 24 do
        for c:=1 to 80 do ar[r,c] := ' ',

        frame (4,4,19,70,4,10,ar) ,
        drawhline (15,15,59,ar) ,
        drawvline (8,9,19,ar) ,

        drawhline (8,4,70,ar) ,
        eraseh     (8,4,70,ar) ,

```



```

drawvline (30,15,18,ar) ;
erasev    (30,15,18,ar) ;

title (2,12,'AAAAAAAAAAAAABBBBBBBBBBBBCCCCCCCCCCCC1234',ar) ;
title (5,5,'aaaaaaa',ar) ; title (5,16,'bbbbbbbbbbb ',ar) ;
title (6,27,'ccccccc', ar) ; title (6,38,'CCCCCcccc',ar) ;
title (7,49,'eeeeeee', ar) ; title (8,60,'ffffffffff ',ar) ;
end.

```

7.5 用表格方式从终端录入数据

采用表格方式从终端录入数据，是一种常用的、更符合于某些使用者习惯的数据录入方式，整个录入过程非常类似于手工填表，简便直观，博得众多用户的喜爱。本节将介绍实现这一录入方式的有关技术。

表格方式录入数据，涉及到本章和前一章介绍的两项技术，即制表、用表和屏幕格式数据入/出技术。要解决的问题，是处理好这两部分内容的正确衔接关系。举个简单的例子。假定我们要在终端屏幕上填写一张发票的内容。发票总是某种确定的表格形式的，在填写之前，应把这张发票显示在屏幕上，包括我们讲制表软件时提到的表格内容、标题和标题栏目名等不需输入的全部内容。在填写发票时，各种要填的内容应按指定的位置、规定的格式填在发票中，这经常是用屏幕格式数据入/出支持软件实现的。可能还有一些数据，是通过对已录入的数据或常量经计算得到的，如发票中录入了单价和数量，总金额为二者之积。这样的数据不应由使用者打入，而是在程序内自动计算，再把结果写到表格中规定的位置并显示在屏幕上。由此，我们可以得出如下结论：

(1) 在以表格方式录入数据之前，必须首先用制表软件制好要用的表格，包括表格标题和各标题栏目名。在录入数据的程序中，需要把表格文件内容显示在屏幕上；也可在程序中直接形成表格。

(2) 在制表过程中，需要规定写入表格中的数据的位置和格式等。其实这些数据正是 I/O 支持软件中相应的 AT 过程的关键参数的值。在我们使用 AT 过程时，可以用编辑程序把这些值写到相应的 AT 的参数位置。如果需要，也可以写一个小程序，让它自动地把数据格式文件（上一节中用到的 FORMAT·FMT 文件）中的有关内容补到用户源程序的恰当位置。这里我们清楚地看到了数据格式文件的作用和具体用法，它是向表格填入数据时进行有关控制的依据。以后我们还会看到，在形成各种报表的过程中，同样要用到这个文件。

下面我们就给出实现填写发票的这个程序例子。程序清单之后给出了程序的运行结果，这是建在磁盘上的名字为 FF·DAT 的一个文件的具体内容。

```

program fapiao (input,output) ;
    (* $include:'interf.pas' *)
var
    form:array [1..24,1..66] of string (2) ;
    lst:array [1..4] of lstring (10) ;

```

```

kar,jar:array [1..4] of lstring (6) ;
itemtotal:array [1..4] of integer;
key,i,j,k,ltotal:integer;
result:text;      title:lstring (30) ;
lst1,lst2,lst3,lst4,itl:lstring (10) ;

```

```
begin
```

```

  upcls (1,1,21,80,0) ; position (1.1) ;
  for i:=1 to 4 do
    [lst [i] := '      ', kar [i] := '      ',
     jar [i] := '      ']; itl:= '      ';
  for i:=1 to 20 do for j:=1 to 40
    do [form [i,j,1] := ''; form [i,j,2] := ''];
  lst1:='名称',      lst2:='单价';
  lst3:='数量',      lst4:='总价';
  title:='清华大学商店发票';      k:=0;

  for i:=2 to 12 do
    for j:=1 to 25 do
      [if i=2 then if j=1 then form [i,j] := 'r' else
        if j=25 then form [i,j] := 'r' else
        if j<>1 and then (j-1) mod 6=0
          then form [i,j] := 'r'
          else form [i,j] := '—' else
        if (i mod 2=0) and (i<>12) then
          if j=1 then form [i,j] := 'r' else
          if j=25 then form [i,j] := 'r' else
          if (j-1) mod 6=0
            then form [i,j] := '+'
            else form [i,j] := '—' else
          if i=12 then if j=1 then form [i,j] := 'r' else
            if j=25 then form [i,j] := 'r' else
            if j<>1 and then (j-1) mod 6=0
              then form [i,j] := '+'
              else form [i,j] := '—' else
            if (j=1) or (j=25) or ( (j-1) mod 6=0)
              then form [i,j] := 'r'
              else form [i,j] := '      '];

```

```

  move1 (adr title [1],adr form [1,5],24) ;

```

```

move1 (adr lst1 [1] ,adr form [3,2] ,6) ,
move1 (adr lst2 [1] ,adr form [3,8 ] ,6) ,
move1 (adr lst3 [1] ,adr form [3,14] ,6) ,
move1 (adr lst4 [1] ,adr form [3,20] ,6) ,
for i:=1 to 12 do
    [for j:=1 to 25 do write (form [i,j] ) , writeln] ,
at (#40,05,32,18,8,ads'F1:单项总价 F5:汇总价') ,
at (#40,05,32,19,8,ads'F10:结束运行 返回监控') ,
repeat
    accept (k) ,
        at (#30,05,6, 5, 5,ads lst [1] ) ,
        at (#20,05,6, 5,17,ads jar [1] ) ,
        at (#20,05,6, 5,29,ads kar [1] ) ,
        at (#30,05,6, 7, 5,ads lst [2] ) ,
        at (#20,05,6, 7,17,ads jar [2] ) ,
        at (#20,05,6, 7,29,ads kar [2] ) ,
        at (#30,05,6, 9, 5,ads lst [3] ) ,
        at (#20,05,6, 9,17,ads jar [3] ) ,
        at (#20,05,6, 9,29,ads kar [3] ) ,
        at (#30,05,6,11, 5,ads lst [4] ) ,
        at (#20,05,6,11,17,ads jar [4] ) ,
        at (#20,05,6,11,29,ads kar [4] ) ,
    keys (0, [1,5,10] ,key) ,
    k:=1,

case key of
1..4:for i:=1 to 4 do
    if (kar [i] <>null) and (jar [i] <>null) and
        decode (kar [i] ,1) and decode (jar [i] ,j)
    then [itemtotal [i] :=1*j,
        position (i*2+3,41) , write (itemtotal [i] :6) ]
    else itemtotal [i] :=0,
5: [total:=0,
    for i:=1 to 4 do total:=total+itemtotal [i] ,
    position (13,20) , write ('汇总价:',total:7) ] ,
10: [assign (result,'ff.dat') , rewrite (result) ,
    for i:=1 to 4 do
        if (lst [i] <>null) and (kar [i] <>null) and (jar [i] <>
            null)
        [ then [move1 (adr lst [i,1] ,adr form [i*2+3,2] ,10) ,

```

```

        move1 (adr jar [i,1] ,adr form [i*2+3,10] ,6) ,
        move1 (adr kar [i,1] ,adr form [i*2+3,16] , 6) ,
        if encode (itl,itemtotal [i] :6)
            then move1 (adr itl [1] ,adr form [i*2+3,20] ,6)
        ] ,
    for i:=1 to 12 do
        [for j:=1 to 25 do write (result,form [i,j] ) ,
        writeln (result) ] ,
        writeln (result,'汇总价:':28,total:7) ,
        close (result) ,
        upcls (1,1,21,80,0) position (1,1) ]
    end
until key=10
end.

```

清华大学商店发票

名 称	单 价	数 量	总 价
a1	2	23	46
QWER	12	23	276
AS	234	12	2808
ASD	1	1	1

汇 总 价: 3131

在这个例子中，表格是在程序中形成的，并用MOVE1过程把表格标题与标题栏目名传入表格相应位置。这之后用 FOR 语句把表格内容显示在屏幕上，并在表格下面给出操作提示信息。

程序后半部分的REPEAT语句的主要功能是：

- (1) 用ACCEPT、AT和KEY三个过程完成发票数据的录入功能；
- (2) 用CASE语句实现计算单项总价、计算发票汇总价和结束程序运行并把操作结果写入文件等不同功能的控制。

这个程序的前半部分，即形成并显示表格的部分，比较容易读懂，与我们前面讲的制表程序的内容非常接近，表格是用ARRAY [1..24,1..66] of string (2) 来定义的，填入表格标题和标题栏目名调用MOVE1过程实现。必须注意正确给出MOVE1过程中form数组的第二个下标值。给出操作提示信息用AT过程实现。这个程序中通过\$INCLUDE编译命令，使用了interf.pas文件，这个文件的内容是给出ACCEPT、AT、KEYS、POSITION和UPCLS等过程的首部并指明它们均为外部过程（它们还要调用用汇编语言实现的几个过程）。

这个程序的后半部分比较难懂。从调用ACCEPT过程到KEYS过程之间的语句，实

现的是录入发票中的数据，用的是在第五章详细讲解的有关知识。录入的数据是商品名、单价和数量，都用LSTRING形式表示。为了计算单项总价，必须用DECODE函数把数字串转换成二进制整数。显示单项总价，是用WRITE语句并配合光标定位功能实现的，单项总价被保存在一个数组中，为以后求发票汇总价提供了方便。

把录入的数据填入表格中，仍然采用MOVE过程，即当商品名不空、单价和数量也不空时，就要把三项内容传入表格的相应位置，并同时把求得的单项总价（整型量）用ENCODE函数转换成数字串形式后传入表格最右一栏中。

对汇总价的处理与对其它数据的处理有些不同，无论将其显示在屏幕上，或写入磁盘文件中，都是用WRITE语句完成的。可以这样做的原因很简单，是因为它接在给出的表格之后，而不是出现在表格之中。

虽然这个程序不是太长，与实用水平尚有一定距离，但它指明的思路和解决问题的关键技术，却是非常有代表性的。读者认真读懂它，就不难写出完全实用的功能类似的程序。

为了简便，我们在给出的程序中未处理诸如建立销售明细帐等有关问题，这超出了本节要讨论的内容范围，它属于业务、财务处理等范畴，与表格方式录入数据的技术并无直接联系。

7.6 报表生成及打印

报表生成，通常是指把一批数据按指定的格式填到确定的表格中。与用表格方式录入数据的区别表现在：

(1) 报表生成，多是以“不可见”方式，由程序自动完成的，而不是由用户从终端上向表格中直接输入数据，即报表中的数据经常是统计性的，是对另外的大量数据进行加工处理得出来的。这种加工处理如果是用计算机完成的，填写报表就不应该留给用户手工进行。这并不绝对排斥一定不能手工录入报表的内容。

(2) 报表的内容，一般要打印出来。可以说，报表和打印的联系是很紧密的。因此，在生成报表时，往往要考虑打印的方便性和实用性。而从屏幕录入数据时，多是录入“原始”数据，又称一次数据，意指未经加工过的数据，往往不直接打印出来。

上述区别也不是非常严格的。例如上一节给出的填写发票的操作，很可能每填完一张就直接打印多份，用于企业内部存查和交一联给购货单位报帐等目的。

下面我们给出了一个把10个人的工资情况填入一张工资报表中的程序例子。该程序是原理性的。它用到四个TEXT类型的文件。fdata对应的实际文件是SALARY.DAT，它里面记录的是10个人的工资情况，每人的记录由职工编号numb、姓名name，和由基本工资jb、工龄工资gl、职务工资zw及奖金jj四部分组成的收入；由房租fz、水费sf、电费df及托儿费tr四部分组成的支出；及实发数等组成。文件中的数值是随意给出的。fdata文件的内容是用编辑程序建立的，不同域的值是通过字符在一行中的位置决定的，应与salary类型定义相吻合，不能错位。

fbiao和ffmt对应的实际文件是baobiao.tbl和baobiao.fmt，是用制表软件形成的表格文件和数据格式文件。后者只给出了部分内容，它指出表格最右下角的行、列值，并指出可以填入表格中的数据的位置，这里只给出了要填入第5行的10个数据的准确位置，以下的9行数据的行值是可计算的，而列值则与第5行中的列值完全相同。fbiao文件的内

容是表格、表格标题和标题栏目名，其内容可以从程序的运行结果中看到。

fout对应的实际文件是gzbiao.dat，保存的是程序运行的结果内容，已给出在程序清单之后。

程序中通过地址类型，把一个string(64)类型的变量str132的前54个字符当作salary类型的数据引用。因为用str132读fdata文件更方便，而选用adrsalrec引用一个人的工资记录的各域非常直观。

该程序比较容易读懂。在完成打开文件之后，接下来的5行语句完成读入表格内容的操作，也就是首先读来表格最右下角的坐标值，再用for语句读表格文件。

再用后面的一小段程序，依次读来10个人的工资记录，并用MOVE过程把所要的数据传送到表格数组的各相应单元。对form数组的第一个下标（即行号）值，用 $i+1 * 2 + 1$ 得到，而第二个下标值，应依次为3、13、23...83，对实发数，form的第二个下标值为93，它们正是ffmt文件中指明的。ffmt文件每一行中的第三个数是一个数据的结束（即最右）位置，我们的程序中未用到它。

接下来的一段程序，用于变换读来的数字串（string(4)类型）为二进制形式的整型变量值，是通过引用本程序中实现的一个函数CONVERT完成的。在该函数中使用了string类型的变量形参，并用UPPER函数求出它下标的上界值，函数执行体内的错误检查，只用于表明实现这种检查的方法，在我们的程序中不会遇到这种错误，除非用该函数变换职工姓名域这种不该出现的用法错误。变换的目的是求出每名职工应发工资数。我们还用ENCODE过程把整型量shifa1变换为LSTRING类型的量shifa2，并将其传送到表格中。

程序的最后部分，把得到的表格内容写到文件fout中，并在屏幕上显示表格前20行中的最左78个字符的实际值。

该程序相对较短，花一点时间认真读懂它是很值得的，以此作为基础，写出功能更强、更加实用的程序是可能的。在此我们未讨论实用中更多的细节问题，请读者自己去思考。

下面是该程序的源清单和有关文件的具体内容。

```
program gzbb (input,output) ;
type
  salary=record
    numb:string (6) ;
    name:string (10) ;
    shouru:record
      jb,gl,zw,jj:string (4)
    end;
    zhichu:record
      fz,cf,df,tr:string (4)
    end;
    shifa:string (6)
  end;
  str132=string (64) ;
```

var

```
adrsalrec:  adr of salary;
adrstr132:  adr of sl32;      str132:sl32;
fdata,fbiao,ffmt,fout:text;  ch:char;
form:packed array [1..30,1..120] of char;
key,row,col,attr,i,j,l:integer;
zw1,jj1,jb1,xl1,fz1,sf1,dfl,trl,shifa1:integer;
shifa2:lstring (10) ;
```

```
function convert (var str:string; var x:integer) :boolean;
var i:integer;
begin
  convert:=true; x:=0;
  for i:=1 to upper (str) do
    if str [i] in ['0'..'9']
      then x:=x*10+ord (str [i] ) - 48
      else if ord (str [i] ) >32 then convert:=false
  end;
```

begin

```
  adrstr132:=adr str132;      adrsalrec.r:=adrstr132.r;
  assign (fbiao,'baobiao.tbl') ;      reset (fbiao) ;
  assign (ffmt,'baobiao.fmt') ;      reset (ffmt) ;
  assign (fdata,'salary.dat') ;      reset (fdata) ;
  assign (fout,'gzbiao.dat') ;      rewrite (fout) ;
```

```
  readln (ffmt,row,col) ;
  for i:=1 to row do
    [for j:=1 to col do
      [read (fbiao,ch) , form [i,j] :=ch] ,
    readln (fbiao) ] ,
```

```
  for i:=1 to 10 do with adrsalrec^,shouru,zhichu do
    [readln (fdata,str132) ,
      row:= (i+1) * 2+1,
      move1 (adr name, adr form [row,3] , 8) ;
      move1 (adr zw,   adr form [row,13] , 4) ;
      move1 (adr jj,   adr form [row,23] , 4) ;
      move1 (adr jb,   adr form [row,33] , 4) ;
      move1 (adr xl,   adr form [row,43] , 4) ;
```

```

        move1 (adr fz,      adr form [row,53] , 4) ;
        move1 (adr sf,      adr form [row,63] , 4) ;
        move1 (adr df,      adr form [row,73] , 4) ;
        move1 (adr tr,      adr form [row,83] , 4) ;

        if convert (zw,zw1) and convert (jj,jj1) and
           convert(jb,jb1) and
           convert (xl,xl1) and convert (fz,fz1) and
           convert (sf,sf1) and
           convert (df,df1) and convert (tr,tr1)
        then [shifa1:=zw1+jj1+jb1+xl1- (fz1+sf1+df1+tr1) ,
              if not encode (shifa2,shifa1:6)
              then writeln ('Data format error')
              else move1 (adr shifa2 [1] ,adr form [row,93] , 6)
              ]
        else writeln ('Input data format error') ] ,
    for i:=1 to 24 do writeln (fout,form [i] ) ,
    for i:=1 to 20 do
        [for j:=1 to 78 do write (form [i,j] ) , writeln] ,
    close (fbiao) , close (ffmt) , close (fdata) ,
    close (fout)
end.

```

SALARY.DAT 文件的内容如下:

1 张大年	45	62	40	5	11	2	4	23
2 李一衣	40	150	40	12	12	2	2	23
3 王必升	29	30	40	5	13	2	1	0
4 赵 昭	140	22	40	12	14	2	1	23
5 天 田	35	0	40	10	15	2	1	0
6 宋颂松	72	12	40	10	16	2	2	12
7 生升胜	60	50	40	10	17	2	4	12
8 大答大	30	30	40	5	17	2	7	0
9 李 犁	112	80	40	5	19	2	9	0
10 卡开刊	57	21	40	5	40	2	8	0

baobiao.fmt文件的内容如下:

```

24 106
5   3 101
5  13 201
5  23 301
5  33 401

```



```

5  43  501
5  53  601
5  63  701
5  73  801
5  83  901
5  93  1001
0  0    0完

```

文件result.dat的内容，即程序的运行结果如下：

工 资 报 表

1988年 8 月

姓 名	基 本 工 资	工 龄 工 资	职 务 工 资	奖 金	房 租	水 费	电 费	托儿费	实发数
张大年	45	92	40	5	11	2	4	23	112
李一衣	40	150	40	12	12	2	2	33	203
王必升	29	30	40	5	13	2	1	0	88
赵 昭	140	22	40	12	14	2	1	23	174
天 田	35	0	40	10	15	2	1	0	67
宋颂松	72	12	40	10	16	2	2	12	102
生升胜	60	50	40	10	17	2	4	12	125
大答大	30	30	40	5	17	2	7	0	79
李 犁	112	80	40	5	19	2	9	0	207
卡开刊	57	21	40	5	40	2	8	0	73

这个程序是用于把已知数据填入确定的表格中。经改动后，是可以得到一个用于处理把任意的文件中所要求的某些数据项，填到指定的任何一张表中的功能的，也就是说，无论有多少种数据结构和多少张表格，只要在它们之间已建立起各自的对应关系，只需用一个通用的程序模块就能处理任何一张报表，而不必为每一张报表单独写一个处理程序。要做到这一点，必须处理好以下几个问题：

(1) 必须用某种方法管理好系统中全部的表格，能建立新表，修改已有表，删除不再有用的表等等，要能方便地指出与找到要用的表格。对一张表，不仅要有表格内容（表格本身，前后标题、各栏目的标题等）文件，还必须同时有记录向表格内写入数据时的数据位置、长度、类型和格式等有关信息的另外一个文件，这就是我们在讲制表软件时反复强调的一个表格的两项主要内容。

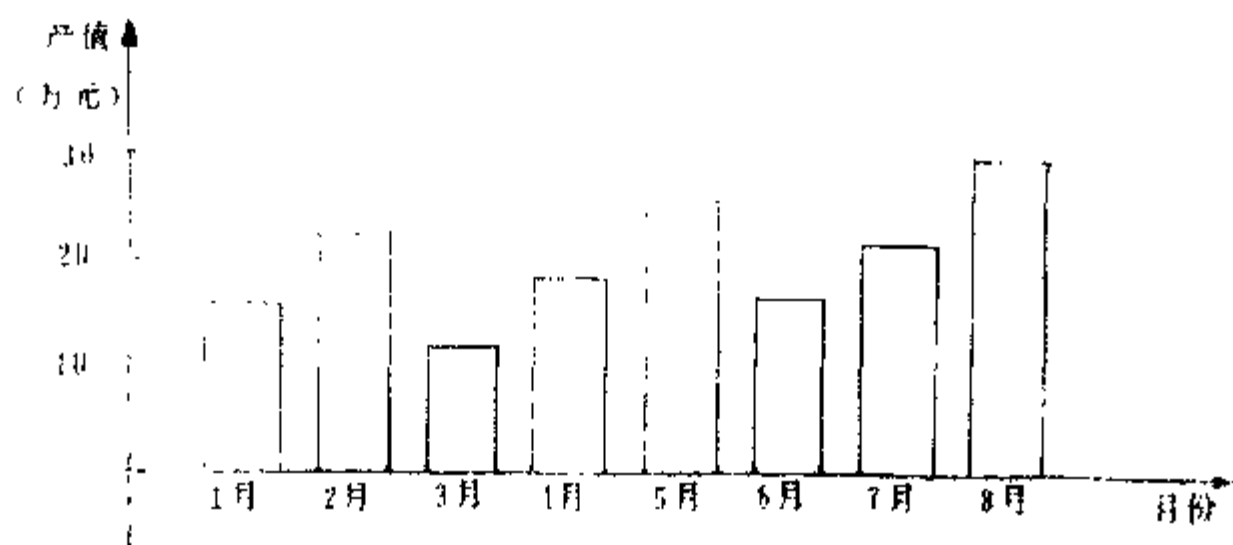
(2) 必须能按用户要求，选择把哪一些数据中的哪几项数据内容（记录的域值）、按什么衔接次序填到指定的表格中。通常的用法，多半是把所要求的数据填到报表中，而不必把某个文件的全部内容填到表格中打印出来，除非该文件中的内容正好是由另外的程序加工处理得到的报表内容。

(3) 报表中所要求的, 往往不像我们前边给出的程序中那样简单, 还有更多的要求。例如, 报表内容很多时, 要打印成多少页, 就有个自动分页、自动加页号的要求。有时还有对某些数据分组、分类统计的要求等等。要处理这些问题, 必须有办法让使用者指明要求, 程序中要有能记下这些要求并按此实现处理的功能。要使报表程序有更好的通用性, 必须解决好这些看起来似乎不怎么重要、而实际上却很杂乱复杂、相当难以全面处理好的实际问题。

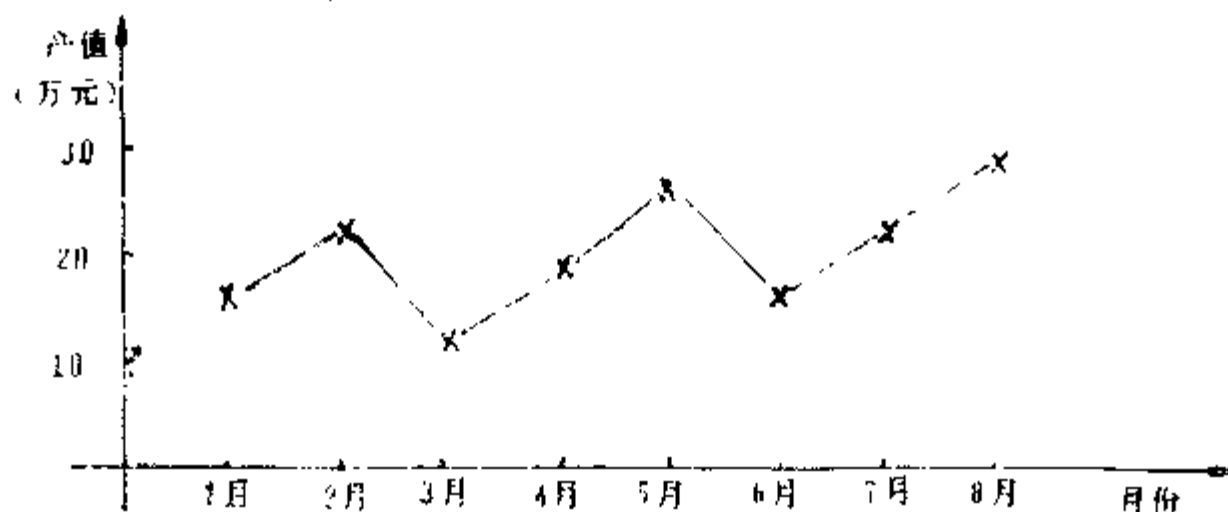
还有其它一些有关联的问题, 我们不在此处进行更深入的讨论。

7.7 实现绘图功能

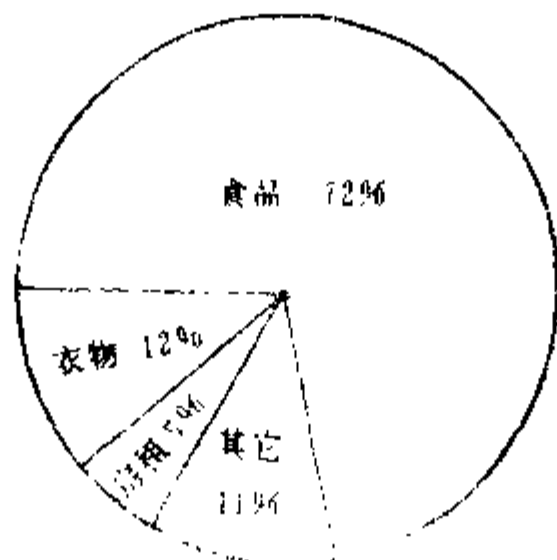
在处理报表的时候, 不仅要用到表格与有关的数据, 有时还要用到各种图形。例如, 用直方图表示一年中前八个月与每个月的生产产值:



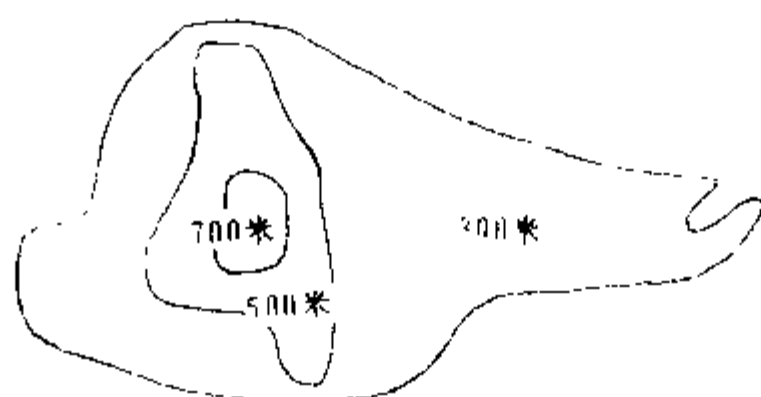
或用折线图表示上图结果:



常用的还有一种圆形图, 经常用于表示某些数据之间的比例关系。例如, 一个家庭某个月的开支情况:



更复杂一点的情况是表示区域，例如地形图上的等高线，天气图中的雨区分布等。此时的图形很不规则。下图是地形图中某一小区域的等高线。



在计算机系统中，怎么绘制出并表示、保存各种图形，是计算机图形学要解决的问题，超出了本书讨论的范围。在本节我们只准备给出绘制上述四种图形的基本原理和具体用法。

要应用图形技术，就要求所用的计算机系统中配置单色或彩色图形终端，并配置图形打印机。大多数的PC机系统所提供的终端和打印机，可以在字符和图形两种方式下运行，并允许由用户加以选择。通常情况下，终端和打印机是在字符方式下运行，在需要绘制或打印图形时，就必须使终端和打印机进入图形方式。在设置终端运行方式时，还必须正确指定终端分辨率。分辨率，是指在终端屏幕上能最多区分出多少个显示点。分辨率越高，显示的字符或图形越清晰。一个终端所能提供的最高分辨率是由硬件决定的，使用时不能超出它的范围，但可以在该范围内的某一较低的分辨率上运行。常用的分辨率有 320×200 、 640×200 、 640×350 等等。也有更高的，例如长城0520的彩色终端多为 900×600 。在字符方式下，多数终端支持每屏显示80字符/行 \times 25行，即2000个字符。也有只支持40字符/行 \times 24行的。

对IBM PC机系统，选择终端运行方式是用INT 10h实现的。调用时，AH放00h，AL放显示方式的参数。当该参数为02h、03h、0Ah和10h时，对应的运行方式分别为80 \times 25黑白字符、80 \times 25彩色字符、640 \times 200的4色图形和640 \times 350的4色或16色（取决于可用RAM）图形。其它参数值的含义请参阅IBM PC机技术资料。为了在PASCAL程序中变换终端方式，要用汇编语言实现一个外部过程，在我们下面给出的例子中，这个过程的名字为DSPMODE。

实现绘图的关键技术，是直接控制终端屏幕上的每一个显示点。要知道，终端（包括电视机）屏幕上所显示的内容都是由发光点给出的。例如字符A、B和?可以分别用如下点阵加以显示：



点之间的距离越小，字形越清晰，用的硬件也就更多、更精细。在字符方式下，当把一个字符送往终端显示时，终端将按收到的字符，从一个被称之为字符发生器的硬件中找出相应的点阵并加以显示，因此，用户不觉得还有什么发光点问题。但在图形方式下，用户就要直接控制这些发光点。用技术术语说，发光点被称为象素（PIXELS，即 PICTURE ELEMENTS）。控制象素包括指定象素位置和象素的显示属性，这可以通过 INT

10h 实现。调用时，DX 寄存器中放行数，CX 寄存器中放列数，AL 中放像素值（显示颜色）。AH 中放子功能码 0ch（16 进制），即 10 进制的 12。为了在 PASCAL 程序中能在终端屏幕上画一个像素点，就要用汇编语言实现一个外部过程，在下面给出的例子中，该外部过程的名字为 PLOTDOT。

在 PASCAL 程序中，上述两个过程要按如下方式说明：

PROCEDURE DSPMODE (W:WORD) ; EXTERN;

PROCEDURE PLOTDOT (ROW,COLUMN,COLOR:INTEGER) ;
EXTERN;

下面给出的是上述两个外部过程的汇编语言的源程序清单。过程的参数 W、ROW、COLUMN 和 COLOR 分别代表显示方式参数、像素的行、列位置和像素值。

```
cseg    segment 'code'
        assume cs:cseg
        public plotdot,dspmode
```

```
plotdot proc    far
        push    bp
        mov     bp,    sp
        mov     dx,    [bp+10]
        mov     cx,    [bp+8]
        mov     ax,    [bp+6]
        mov     ah,    0ch
        int     10h
        pop     bp
        ret     6
```

plotdot endp

```
dspmode proc    far
        push    bp
        mov     bp,    sp
        mov     ax,    [bp+6]
        xor     ah,    ah
        int     10h
        pop     bp
        ret     2
```

dspmode endp

```
cseg    ends
        end
```

在 PASCAL 程序中实现绘图，是向图形应出现的位置上的每一个像素执行写操作，这个写操作是通过调用外部过程 PLOTDOT 完成的。例如，为了在终端屏幕的 100 行上从

50列到600列的位置画一条横线，可用如下语句完成：

```
for c:=100 to 600 do plotdot (100,c,2)
```

这里说的行、列值都是指图形方式下的象素位置。假定彩色终端的分辨率为 640×350 ，则表明终端在水平方向每行由640个象素组成，垂直方向每列由350个象素组成。此时 plotdot 过程的第一个参数的取值范围在0到349之间，第二个参数的取值范围在0到639之间。终端最左上角的那个象素的坐标值为(0, 0)，最右下角的那个象素的坐标值为(349, 639)。过程的第三个参数的取值范围在0到3、或0..7之间，取决于设备硬件。

绘图是要写出用到的每一个象素的。涉及到的象素可能很多，前面刚提到的画一条水平线的例子，就是调用501次plotdot过程，处理不当，绘图的速度会明显降低。我们不详细说明提高绘图速度的方案，只准备提两点：

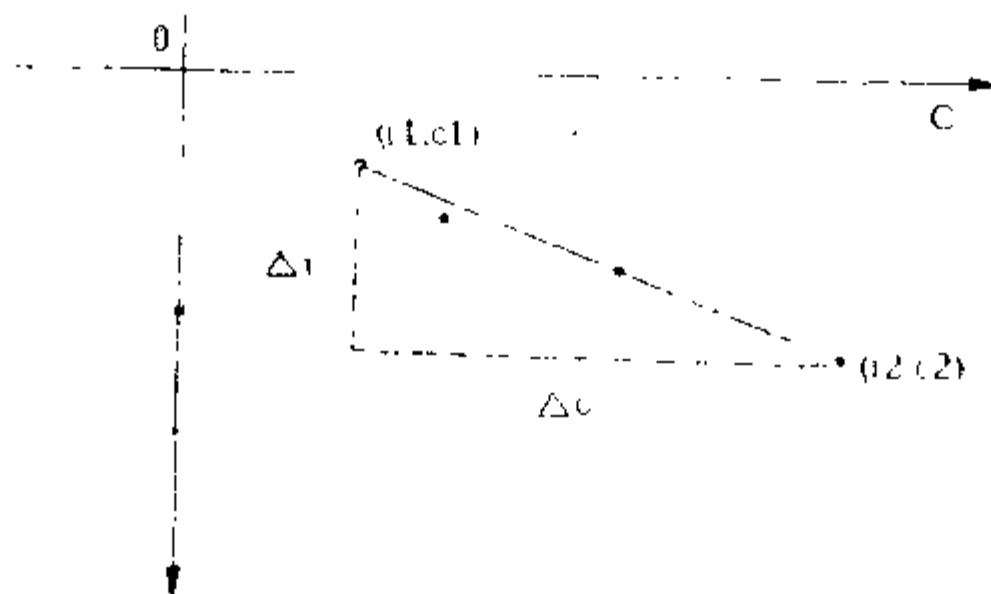
(1) 从所用的语言考虑，用汇编语言实现绘图，是提高绘图速度的重要途径，例如要完成动画操作，不用汇编语言很可能达不到预期的效果；在本书讨论的应用领域中，这个矛盾一般不很突出。

(2) 从所用的算法考虑，要尽量避免使用复杂的计算，如开方、乘法、除法，要尽量避免使用实数运算和各种函数，要尽量减少不必要的重复计算。这是因为绘图要重复执行很多语句，哪一处速度慢一点，多次重复后造成的速度下降就非常明显。我们准备结合绘制斜线和圆所用的算法来说明这一问题。

绘制一条水平线或垂直线是相当简单的，用for语句控制在水平或垂直方向上每个象素坐标值的变化，让过程的另一个参数值保持不变，多次调用PLOTDOT过程就实现了。而绘制斜线，行和列的值都要变化，情况就复杂一些。例如，要从坐标位置(r_1, c_1)到(r_2, c_2)画一条线，可用如下语句完成：

```
for c:=c1 to c2 do  
  [r:=r1+round((c-c1)*(r2-r1)/(c2-c1));  
   plotdot(r,c,1)];
```

这在原理上是完全正确的，但在计算每个r值时，都用到了乘法、除法和舍入函数round，在循环语句内，一些常量还被反复计算，这大大降低了绘制斜线的速度。可以用另外一种算法高速绘制斜线，这种算法就是著名的Bresenham算法。



Bresenham 算法的实质是跟踪一个误差项，用整数的加减法取代乘法、浮点除法与浮点数舍入取整等操作。开始时，让误差项取值为0，然后每当c的值增1，就通过比较

$\Delta c \text{ DIV } 2$ 的值和误差项与 Δr 的差值，并依据比较结果决定是否要使 r 增1和按 Δc 修改误差项。这对于计算得到的 r 值在理论计算线的上方、下方与正好在线上都是正确的。

这里还有两个问题要说明。一是 r 和 c 的变化方向，二是 r 和 c ，哪一个作自变量，哪一个由计算得到。对第一个问题，当线的终点值的 r_2, c_2 分别大于原点值的 r_1, c_1 时， r 和 c 都按增1方向变化。否则， $r_2 < r_1$ ， r 按减1方向变化， $c_2 < c_1$ ， c 按减1方向变化。在程序中， r 是按增1还是按减1方向变化，记在BR变量中，BC记录的是 c 的变化方向。对第二个问题，应该按 Δr 和 Δc 哪一个值大来决定 r 和 c 中哪一个作为自变量，当 $\Delta r > \Delta c$ 时，要选 r 作自变量，否则画出的点数不够，请看下图：



r 作自变量，4个点画全了，当用 c 作自变量时，只画出两个点。各相邻像素之间横向、竖向的距离，在图上都有意拉大了，以便看得更清楚。

用Bresenham算法实现的绘制斜线的程序，被设计成一个过程line。水平线和垂直线均为斜线的特例，line过程也能用来绘制水平线（此时 $r_1 = r_2$ ），也能用来绘制垂直线（此时 $c_1 = c_2$ ）。在line过程中，画线是用WHILE语句结构实现的，选择自变量和确定 r 和 c 的变化方向的办法，从程序中可以看得很清楚。

在程序中，还通过给出左上角和右下角的像素坐标值，四次调用line过程实现的绘制一个矩形的过程SQUARE。

下面说明绘制一个圆的具体算法。画圆是用一个过程PLOT CYCLE实现的，它的参数包括圆心的坐标 $center_r, center_c$ ，圆的半径 rr ，画圆选用的颜色值 CL 。画圆时，要按 $rr^2 = r^2 + c^2$ 的关系，取 r 或 c 作自变量，计算 c 或 r 的值。在实现中，实际上只要计算圆周的1/8的坐标点的 r 和 c 的值就可以了。我们计算的是圆在直角坐标的第四个象限靠近 x 轴处的圆周每一点的坐标值，把 r 作为数组的下标值， c 作为数组元素的内容加以记忆。然后用八个for语句画出一个圆周的八个部分。请注意，计算圆周每个点的坐标值时，用的只是乘法和减法，避免了使用乘方，平方和实数的舍入取整等函数运算，这对加速计算过程是有利的。

下面给出的是实现绘图的一个PASCAL程序源清单。

程序中加了清楚的注释，具体功能可以从注释中看到。

我们还可以把该程序中实现的几个过程，用一个MODULE实现出来，这样一来，在任何需要实现绘图的程序中，通过调用这些过程就能完成绘图功能。

```
program plotgraphe (input,output);
var ch:char; r,c,cl:integer;
    ar:array [0..120] of integer;

procedure plotdot (row,column,color:integer); extern;
```

```

    procedure dspmode (w:word) ; extern;

    procedure line (r1,c1,r2,c2,c1:integer) ;
var delta-r, delta-c, halfr, halfc, br, bc, errorterm:integer;
begin
    br:=1;    bc:=1;
    delta-r:=r2-r1;
    if delta-r<0 then [br:=-1; delta-r:=-delta-r] ;
    delta-c:=c2-c1;
    if delta-c<0 then [bc:=-1; delta-c:=-delta-c] ;
    halfr :=delta-r div 2;    halfc:=delta-c div 2;

    r:=r1;    c:=c1;    errorterm:=0;
    while (r< >r2) or (c< >c2) do
        [plotdot (r,c,c1) ;
        if delta-r>delta-c
            then [if r< >r2 then r:=r+br;
                errorterm:=errorterm+delta-c;
                if errorterm>halfr
                    then [errorterm:=errorterm-delta-r;
                        c:=c+bc] ]
            else [if c< >c2 then c:=c+bc;
                errorterm:=errorterm+delta-r;
                if errorterm>halfc
                    then [errorterm:=errorterm-delta-c;
                        r:=r+br] ]
        ] ; plotdot (r2,c2,c1)
    end;

    procedure square (r1,c1,r2,c2,c1:integer) ;
begin
    line (r1,c1,r1,c2,c1) ;
    line (r1,c2,r2,c2,c1) ;
    line (r2,c2,r2,c1,c1) ;
    line (r2,c1,r1,c1,c1)
end;

    procedure plotcycle (centerr,centerc,rr,c1:integer) ;
var c1:integer;
begin
    c1:=rr*7 div 10+rr*7 div 1000;    c:=rr;

```

```

for r:=0 to cl do
    [if r*r + (c-1) * (c-1) >= rr*rr then c:=c-1; ar[r] := c];
for r:= 0 to    cl do plotdot(r+centerr,ar[r] +centerc,cl);
for c:=cl downto0 do plotdot(ar[c] +centerr,c+centerc,cl);
for c:= 0 to    cl do plotdot(ar[c] +centerr,centerc-c,cl);
for r:=cl downto0 do plotdot(r+centerr,centerc-ar[r],cl);
for r:= 0 to    cl do plotdot(centerr-r,centerc-ar[r],cl);
for c:=cl downto0 do plotdot(centerr-ar[c],centerc-c,cl);
for c:= 0 to    cl do plotdot(centerr-ar[c],c+centerc,cl);
for r:=cl downto0 do plotdot(centerr-r,ar[r] +centerc,cl)
end;

begin
    write(chr(27),'[j');          ! 清屏
    dspmode(#10);                 ! 置终端为图形方式
    r:=0;
    while r<=350 do                ! 画若干条横线
        [for c:=0 to 639 do plotdot(r,c,1); r:=r+30];

    c:=0;
    while c<=639 do                ! 画若干条竖线
        [for r:=0 to 349 do plotdot(r,c,1); c:=c+40];

    for r:=0 to 349 do plotdot(r,r,2); ! 画一条斜线
    line(200,50, 0,630,2);          ! 画一条斜线
    line(0, 0, 0,400,2);            ! 画一条横线
    line(0, 0,200, 0,2);            ! 画一条竖线

    square(110,210,310,610,3);      ! 画一个矩形

    plotcycle(170,300,160,3);        ! 画一个圆形

    writeln(chr(27),'[10, 20H Enter any key to exit');
    read(ch);
    dspmode(#03)                    ! 置终端为80×25彩色字符方式
end.

```

下面给出几个应用图形的程序例子。假定某工厂1987年的产值为：
 第一季度为180万元
 第二季度为150万元
 第三季度为100万元

第四季度为250万元

我们用直方图、折线图和圆形图来表示该厂这四个季度的生产产值情况。

对直方图，主要问题是在解决屏幕上合理的布局，使显示的直方图在水平方向和垂直方向上匀称的、较成比例的占用整个屏幕。此外，还可以把得到的直方图内填实为某种颜色，以及给出横轴竖轴的坐标标记和坐标值。

对折线图，也必须合理地在屏幕上布局，并正确地给出折线的折转点坐标位置。

对圆形图，主要是计算比例分配，即四个季度中，每个季度的产值占全年总产值的百分比，并把这个百分比变为圆形图中的一个扇形区。这实际上是用计算圆周上的一个坐标点，并画一条从圆心到该点的斜线的办法实现的。此外，也存在把一个扇区填实为某种颜色的要求。而在扇区内填颜色，比在直方图中填颜色要复杂得多。为此，通常要写一个在任何封闭图形内填颜色的过程。实现的办法，是从封闭图形内的任何一点，选用某种方案逐步向四周扩展。先读一个像素的现行值，发现其值为零（不是封闭图形的外框线上的一个点），则按要求的像素值执行写操作，不为零，表明到达了封闭图形外框上的某一点。

第一个程序，实现用直方图在屏幕上显示四个季度产值的变动情况。

该程序用到几个外部过程，其说明写在‘BB.INF’文件内，并通过\$INCLUDE编译命令将其引入到程序中。

程序中用到的另外一个外部过程，实现定位光标到指定的屏幕位置，X为列号，Y为行号，PAGE代表屏幕页，通常为零。

该程序首先设置终端为640×350的彩色图形工作方式。接下来的两个语句绘制出图形的横竖坐标轴。在给出每个季度的产值数据并将其记在数组A中之后，用一个FOR语句，把四个季度的产值变成一定大小的长方条图形显示在屏幕上。这之后的两个语句在竖坐标轴上写上单位标记。接下来的六行程序语句，用于写出横竖坐标轴上的坐标值和坐标的单位。

READLN (CH) 语句等待用户打入一个字符，以便看清显示在屏幕上的图形。最后一个语句用于恢复终端为80×25彩色字符工作方式。在切换终端工作方式（图形→字符，或字符→图形）时，将引起一次清屏操作。

下面是该程序的清单。

```
program ex1 (input,output) ;
var
  j,i,x,y: integer;
  ch: char;
  a:array [1..4] of integer;
  (* *include: 'bb.INF' *)

  procedure putcurs (y,x: integer; page: word) ; extern;
begin
  write (chr (27) , ' [J' ) ;
  dspmode (#10) ;
  line (300,60,300,340,5) ;
  line (300,60,20,60,5) ;
```

```

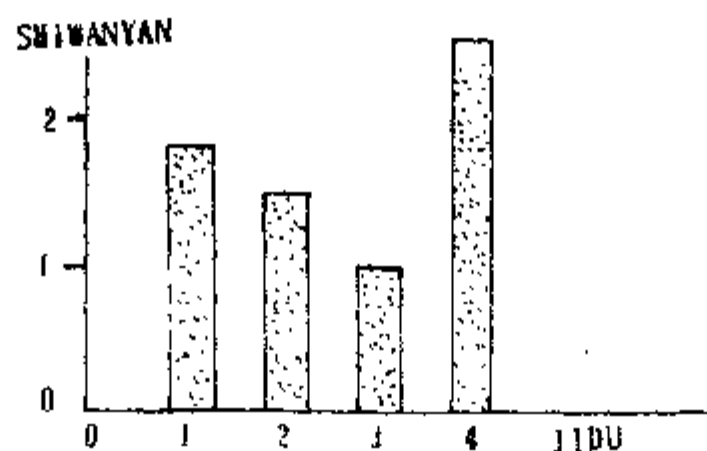
a [1] := 180;      a [2] := 150;
a [3] := 100;      a [4] := 250;
for i:=1 to 4 do
    [x:=50*i,      y:=a [i] ;
    for j:=0 to 20 do
        line (300,x+j+60,300-y.x+j+60,3) ] ;
    line (200,55,200,62,6) ;
    line (100,55,100,62,6) ;

    putcurs (22,7,0) ;
    write ('0 1 2 3 4 JIDU') ;
    putcurs (21,4,0) ; write ('0') ;
    putcurs (14,4,0) ; write ('1') ;
    putcurs ( 7,4,0) ; write ('2') ;
    putcurs ( 3,1,0) ; write ('BAIWANYAN') ;

    readln (ch) ;
    dspmode (#03)
end.

```

下面是程序的运行结果:



第二个程序，实现在终端屏幕上用折线图显示一年四个季度产值变动情况。下面是该程序的清单和运行结果。在这个程序中，没有给出横竖坐标的单位及标记等。

```

program ex2 (input,output) ;
var
    j,k,i,x,y:integer;
    ch:char;
    a:array [1..4] of integer;
    (* ⅲinclude:'bb.pas' *)

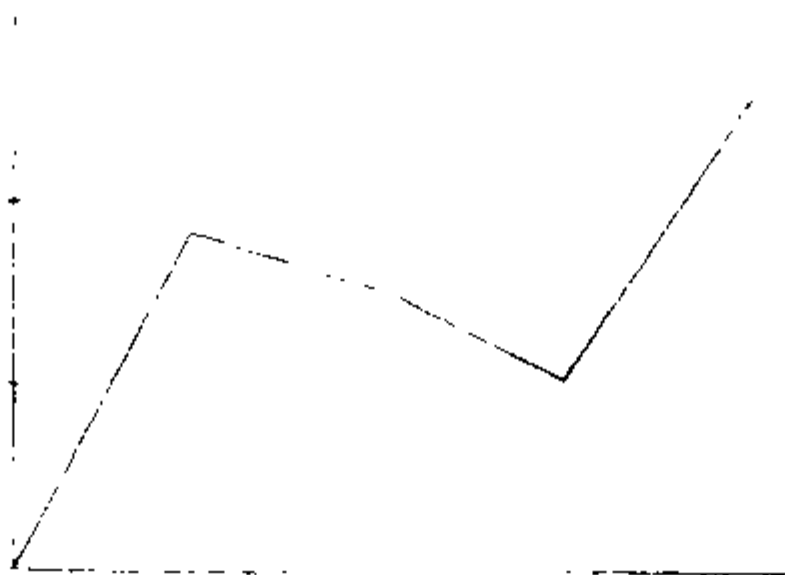
begin
    write (chr (27) , ' [J') ;
    dspmode (#10) ;

```

```

line (300,20,300,300,5) ;
line (300,20, 20, 20,5) ;
a [1] :=180; a [2] :=150;
a [3] :=100; a [4] :=250;
j:=300; k:=20;
for i:=1 to 4 do
    [x:=50*i; y:=a [i] ;
    line (j,k,300-y,x,3) ;
    j:=300-y; k:=x] ;
readln (ch) ;
dspmode (#03)
end.

```



第三个程序，实现用圆形图表示一年四个季度产值的比例情况。与第二个程序类似的是没有给出横竖坐标及其上面的单位标记，也没有在圆形图中给出每个季度的产值，而要实现这一切并不难，与第一个程序中的做法差不多。下面是该程序的清单及运行结果。

```

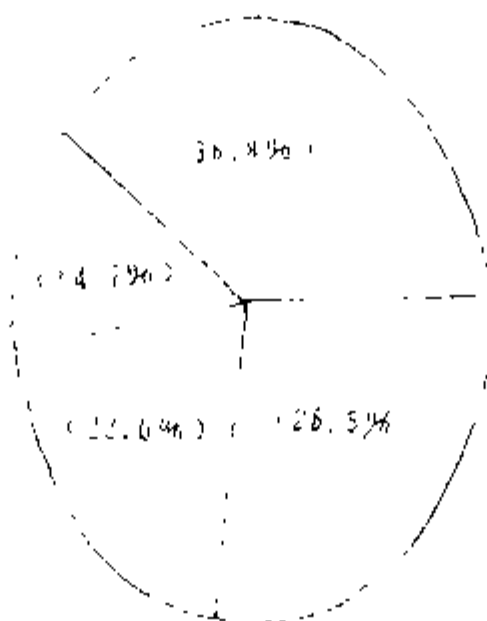
program ex3 (input,output) ;
var
    i,x,y:integer;
    r:real;
    ch:char;
    a:array [1..4] of integer;
    (* *include: 'bb.pas' *)
begin
    write (chr (27) , ' [J') ;
    dspmode (#'Q') ;
    a [1] :=180; a [2] :=a [1] +150;
    a [3] :=a [2] +100; a [4] :=a [3] +250;
    for i:=1 to 4 do
        [r:=a [i] / a [4] * 2 * 3.1415926;
        x:=300+trunc (160*cos (r) ) ;

```

```

        y:=170+trunc (160* sin (r) ) ;
        line (170,300,y,x,3) ] ,
plotcycle (170,300,160,6) ;
readln (ch) ;
dspmode (#03)
end.

```



这个程序是按画圆绘制的，但在屏幕上显示出来，却是椭圆形状，这是由于终端屏幕在横向与纵向像素的密度不完全相等造成的。如果想得到真正像圆一样的图形，可以对横向和纵向坐标值按适当比例处理一下。实用中可以在图中给出每个季度的产值占全年产值的比例，如图上在括号中给出的数值。

7.8 打印机的打印控制

PC 机系统中配置的打印机，多为点阵行式打印机。这类打印机成本比较低，通用性强，一般都兼有打印字符、图形和汉字的功能，只是打印速度稍慢一点。

在使用这类打印机时，除正确设置初始参数和进行正确的连接之外，还必须了解打印机提供的、能在程序中使用的控制命令，以完成打印机工作方式(字符还是图形)的切换，各国专用字符的选择，字体选择，打印时的行间距和字符密度选择，打印速度选择等等。不同厂家的、不同型号的打印机提供的命令，从命令数目、格式和功能等都有很大差异，在使用时必须参阅有关手册。下面我们以后日本EPSON公司生产的 LQ-1000K 打印机为例来说明这些命令的格式和功能，并用一些小程序介绍某些命令的具体用法。如果所用的不是这种型号的打印机，这里的内容仍有一定参考价值。实用中必须按所用的打印机的具体说明，正确选用各有关命令，而不能简单照抄这里的程序实例。

LQ-1000K 是能直接打印汉字的高质量24针中文打印机，内部配有一个汉字 ROM (7547个汉字)，并允许用户自行增设94个附加符号或汉字。每行打印宽度最大为 136 个字符。

一、命令综述

LQ-1000K提供的命令，从命令格式上分，包括如下三类：

(1) 控制码, 由 1 个字符的控制码表示的命令, 例如: CHR (12) 完成跳页; CHR (13) 完成回行; CHR (24) 完成消除; CHR (127) 完成删除字符几项功能。

(2) 由 ESC 开头给出的控制序列

ESC, 即 CHR (27) 为 ASCII 码的换码标志, 后面可以跟上一到多个字符或某些控制码 (其 ASCII 值小于 32) 形成的控制命令。

(3) 由 FS, 即 CHR (28) (为 ASCII 码的文件分隔符) 开头, 后面跟上某些字符形成的控制命令。

从命令的功能区分, 包括如下六类:

- 打印机操作命令
- 数据控制命令
- 打印中文命令
- 加强印字命令
- 选择字符组 (西文字形) 命令
- 映象图形命令

下面我们按其功能, 分类给出命令的格式和功能。

(1) 打印机操作命令

命令格式	命令功能
ESC @	恢复打印机到初始设定状态
ESC <	回车
CHR (7)	响铃
CHR (17)	打印机接收数据
CHR (19)	打印机停止接收数据
ESC $s \begin{smallmatrix} 1 \\ 0 \end{smallmatrix}$	半速打印 } 选择打印速度 全速打印 }
ESC $U \begin{smallmatrix} 1 \\ 0 \end{smallmatrix}$	单向打印 } 选择打印单/双方向 双向打印 }

(2) 数据控制命令

CHR (13)	开始打印缓冲区内容, 印完后回车
CHR (24)	取消要打印的一行数据
CHR (127)	若其前边有一个未打印的字符, 删掉它不打

(3) 打印头移动控制

CHR (12)	打印缓冲区内容, 打完后跳页
CHR (10)	印完缓冲区内容后进到下一行
CHR (8)	退格
ESC N n	设置页的底边界, n 为 1~127 之间的行数
ESC O	消除页的底边界
ESC C CHR(0) n	设置页的长度为 n 英寸, n 的取值范围在 1~22 之间
ESC C n	设置每页的长度为 n 行, n 的取值范围在 1~127 之间

ESC	0		选择行间距为1/8英寸
ESC	2		选择行间距为1/6英寸
ESC	3	n	选择行间距为n/180英寸, n的取值范围在0~255之间
ESC	A	n	设定向后跳行命令的行间距为n/60英寸 (0 < n < 85)
ESC	J	n	将纸往前送n/180英寸 (0 < n < 255)
ESC	B	n1 n2...CHR(0)	以当前行间距设定最多16个垂直跳格位置, 绝对跳格位置不受以后行间距变更的影响, n1、n2...n16依次在1~255范围内逐渐增加。

NUL码, 即CHR(0)表示命令结束。

ESC B CHR(0)用于清除原来的跳格设定。

ESC l n 设定左边界在当前字符宽度的第n列位置, n的取值在1到254之间, 最多可到8英寸。

ESC Q n 设定右边界在当前字符宽度的第n列位置。n的取值在2到254之间。

ESC \$ n1 n2 移打印头到一个绝对点位置, 点的位置由(n2 * 256 + n1)的值决定, 最大为815。点距为1/60英寸, n1的值在1到254之间, n2的取值应大于0小于3。

ESC \ n1 n2 移打字头到距当前位置有一定相对距离的点位置, 这一相对距离的值由公式n2 * 256 + n1决定。这个命令与ESC \$实现的按绝对点位置移动不同。该命令中的点距取决于当前字符的大小和字体选择。当字体为印刷体时, 点距为1/180英寸, 为普通字体时, 点距为1/120英寸。

CHR(11) 垂直跳格

CHR(9) 水平跳格

ESC D n1 n2...CHR(0)

此命令最多允许设定32个水平跳格位置, n1, n2, ... n32在1到255范围内依次增加, CHR(0)表示结束。初始设定值为每次增8。

(4) 字型、字体选择控制命令

ESC	x	0	普通字体 (draft)
		1	印刷字体 (letter quality)
ESC	P		选择10 CPI (每英寸10个字符), 这是初始设定状态。
ESC	M		选择12 CPI (每英寸12个字符)
ESC	g		选择15 CPI (每英寸15个字符)
ESC	p	1	进入比例印字状态
		0	退出比例印字状态

比例印字状态是指按10 CPI的大小打印每个字符,但每个字左右的距离位置与字符本身的宽窄成比例,使打印出的文字文章更漂亮。

CHR(15) 选择缩小的间距状态,使 ASCII 字符印出的字符宽度为标准宽度的60%,本状态下,比例印字状态将无效

ESC CHR (15) 同上

CHR (18) 取消缩小的间距状态

CHR (14) 选择双倍宽度打印方式,即使每个 ASCII 字符的打印宽度加倍,它只对该命令所在的一行有效。

ESC CHR (14) 同上

CHR (20) 取消由 CHR (14) 或 ESC CHR (14) 设定的双倍宽度打印方式

ESC W¹₀ 进入双倍宽度打印状态

脱离双倍宽度打印状态

(5) 黑体字或特定方式打印控制命令

ESC E 打印黑体字,即每点打印两次,第二次打印在第一次打印的稍右位置,印出黑体字,打印速度要慢一些。

ESC F 取消由 ESC E 命令设置的黑体字打印状态

ESC G 打印黑体字,每一行都打印两次,且第二次打印稍低于第一次打印位置,此状态又称双重撞击。

ESC H 取消由 ESC G 命令设置的黑体字打印状态。

ESC E 和 ESC G 命令可复合使用,使打印出的字更重更黑。

ESC S⁰₁ 进入上标打印状态

进入下标打印状态

该项命令是被用于完成上标或下标打印,即使命令后的字符以通常高度的 2/3 左右印出,并且其位置上移或下移半个字符高度。

ESC T 取消由 ESC S 给出的上标或下标打印状态

ESC —¹₀ 进入划底线状态

取消划底线状态

此命令在打出字符的下边再划一条横线。

ESC a n 选择校正状态

n=0 选左边对齐方式打印 (初始设定)

n=1 选中间对齐方式打印

n=2 选右边对齐方式打印

n=3 选满篇打印方式

ESC CHR (32) n 增加字符间间距,即在原间距之外,再在每个字符右侧增加 n 点的空格位置 ($0 < n < 127$),初始设定状态 $n = 0$ 。

(6) 字符组选择命令

ESC 4 选择斜体字状态

ESC 5 消除斜体字状态

ESC R n 选择国籍文字组

当 n 为 0 ~ 12 时,选择的国家依次为美、法、德、英、丹麦 I、瑞典、意大利、西班牙

牙 I、日本、挪威、丹麦 I、西班牙 I、拉丁美洲。

(7) 汉字打印的控制命令

FS &	进入汉字打印状态
FS .	退出汉字打印状态
FS CHR (14)	设定单行双倍宽度汉字打印状态
FS CHR (20)	取消单行双倍宽度汉字打印状态
FS W 1	设定双宽双高汉字打印状态
FS W 0	取消双宽双高汉字打印状态
FS J	设定垂直打印汉字状态, 即把其后的每个汉字旋转90°打印出来
FS K	取消垂直打印汉字的状态
FS — n	选择汉字打印中的划底线功能
n=0	取消划底线功能
n=1	设定划一个点宽的底线状态
n=2	设定划2个点宽的底线状态
FS D	在垂直打印汉字时, 用此命令设置在一个中文位置上印两个 ASCII 字符
FS S n1 n2	设置中文文字间的间距, n1 决定汉字左边的间距, n2 决定汉字右边的间距, n1和n2应在 0~126 范围内, 初始设定为 n1=0, n2=3, 所用单位为1/180英寸。
FS x 1	进入高速打印汉字状态
FS x 0	取消高速打印汉字的状态
FS 2 a1 a2 ...数据...	自行定义汉字

用此命令, 使用人员可以自行定义最多 94 个打印机中原来没有的汉字。其中 a1 为第一个字节, 其值必须为 #F8, a2 为第 2 个字节, 其值必须为 #A1 到 #FE 的范围内。数据是定义的汉字的点阵, 每个汉字由 24×3 个字节组成, 具体用法请参阅打印机手册。

(8) 打印图形的控制命令

ESC K n1 n2 ...数据...

选择 8 位元单密度的图形状态。假定 d 为图形所用的总列数, 则 $n1 = d \text{ MOD } 256$, $n2 = d \text{ DIV } 256$ 。数据由 d 组八位的二进制数组成。

ESC L n1 n2 ...数据...

选择 8 位元双倍密度的图形状态。

ESC Y n1 n2 ...数据...

选择 8 位元高速度双倍密度的图形状态

ESC Z n1 n2 ...数据...

选择 8 位元四倍密度的图形状态

ESC * m n1 n2 ...数据...

图形打印综合的命令, 此处的 d、n1 和 n2 的含义和 ESC K 命令中介绍的相同, 用 m 的值来选择一种图形打印状态, 其关系如下表所示:

m	状 态 名	转 换 码	点 密 度	所用针数	邻 接 点
0	单密度	ESC K	60	8	可能
1	双倍密度	ESC L	120	8	可能
2	高速双倍密度	ESC Y	120	8	不能
3	四倍密度	ESC Z	240	8	不能
4	CRT图形	无	80	8	可能
6	CRT图形 I	无	90	8	可能
32	单密度	无	60	24	可能
33	双倍密度	无	120	24	可能
38	CRT图形 II	无	90	24	可能
39	三倍密度	无	180	24	可能
40	六倍密度	无	360	24	不能

点密度是指每英寸长度中打印的点数。

二、命令应用实例

下面给出五个应用打印机控制码或控制命令的小程序。

第一和第二个程序，表明怎样控制打印汉字。打印汉字的有关命令，都是由FS (CHR (28)) 开头的。第一个程序，主要表明怎样正常打印汉字、垂直打印汉字、双宽打印汉字、双宽双高打印汉字、垂直双宽双高打印汉字等。第二个程序主要表明怎样打双黑体汉字、带细下划线的汉字、带稍粗下划线的汉字、带右划线的垂直打印的汉字，以及怎样变更汉字之间的间隔距离，怎样在垂直打印汉字的情况下，在一个汉字位置上打印两个普通字符的有关方法。

下面是第一个程序的源清单和它的运行结果。

```

program hanzi (input,output) ;
var f:text; esc,fs:char;
    lst:lstring (20) ;
begin
    esc:=chr (27) ; fs:=chr (28) ;
    assign (f,'lpt1:'); rewrite (f) ;
    lst:='打印汉字'; write (f,esc,'@') ;
    writeln (f,fs,'&',lst) ;
    writeln (f,fs,'J',lst,fs,'K') ;

```

```

writeln (f,fs,chr (14) ,lst,fs,chr (20) ) ;
writeln (f,fs,'W!',lst) ;
writeln (f,fs,'J',lst,fs,'K') ;
writeln (f,fs,'WO',lst) ;
close (f)
end,

```

打印汉字

打印汉字

打印汉字

打印汉字

打印汉字

打印汉字

下面是第二个程序的源清单和它的运行结果。

```

program hanzi (input,output) ;
var f:text; esc,fs:char;
    lst:string (20) ; i:integer;
begin
    esc:=chr (27) ; fs:=chr (28) ;
    assign (f,'lpt1:'); rewrite (f) ;
    lst:='打印汉字'; write (f,esc,'@')
    write (f,fs,'&') ;
    writeln (f,lst) ;
    writeln (f,esc,'G',lst,esc'H') ;
    writeln (f,fs,'-',chr (1) ,lst) ;
    writeln (f,fs,'-',chr (2) ,lst) ;
    writeln (f,fs,'J',fs,'-',chr (1) ,lst,fs,'K') ;
    write (f,fs,'-',chr (0) ) ;
    writeln (f) ;
    for i:=0 to 10 do
        writeln (f,fs,'s',chr (2) ,chr (2*i) ,lst) ;
        write (f,fs,'S',chr (0) ,chr (3) ) ;
        writeln (f,fs,'J','1209+@o') ;
        writeln (f,fs,'D','1209+@o',fs,'K') ;
        write (f,chr (28) ,'.') ;
    close (f)
end

```

end.

打印汉字
打印汉字
打印汉字
打印汉字
打印汉字

打印汉字
打印汉字
打印汉字
打印汉字
打印汉字
打印汉字
打印汉字
打印汉字
打印汉字
打印汉字
- 20 < 100
20 < 100

第三个程序，表明怎样打印各种不同的西文字符。正常打印，以印刷体打印，黑体方式打印，更黑字体打印，带下划线打印、斜体方式打印、下脚标打印和上标打印等等。

下面是该程序的源清单和它的运行结果。请特别注意结果的最后一行中的 A_{10} 和 b^2 。

```
program xiwen (input,output) ;
var f:text; es0,fs:char;

begin
    esc:=chr (27) ;          fs:=chr (28) ;
    assign (f,'lpt1:') ;      rewrite (f) ;
    writeln (f,fs,' ', 'printing some contents') ;
    writeln (f,esc,'x1', 'letter quality,') ;
    writeln (f,esc,'E', 'with emphasized printing,') ;
    writeln (f,esc,'F',esc,'G', 'with double strike printing,') ;
    writeln (f,esc,'E', 'and with both effects,') ;
    write (f,esc,'@') ;

    writeln (f,esc,'-1', 'This is underlined by the printer', esc,'-0') ;
    writeln (f,esc,'4', 'This is italic printing',esc,'5') ;
    writeln (f,'x123=A',esc,'S1', '10',esc,'T',
              '+b',esc,'SO', '2',esc,'T', ' -100') ;
    close (1)

end.
```

```

printing some contents
Letter quality,
with emphasized printing,
with double-strike printing,
and with both effects.
This is under lined by the printer
This is italic printing
 $\times 123 = A_1 + b^2 - 100$ 

```

第四个程序将表明怎样运用自造一个汉字的方法，来造一个可供打印机打印的小闹钟形状的符号，并在打印机上用打印汉字的办法，把这一符号连同几个字母一同打印出来。

下面是该程序的源清单和它的运行结果。

```

program hanzi (input,output) ;
type arr=packed array [1..24*3] of byte;
const ar=arr (  #00, #00, #00,      #00, #00, #00,      #07, #c0, #00,
                 #28, #2e, #00,      #10, #71, #80,      #20, #c0, #41,
                 #41, #40, #43,      #42, #80, #27,      #46, #80, #2d,
                 #49, #00, #98,      #51, #01, #10,      #21, #06, #10,
                 #03, #7c, #10,      #21, #00, #10,      #51, #00, #10,
                 #49, #00, #18,      #46, #80, #2d,      #42, #80, #27,
                 #41, #40, #43,      #20, #c0, #41,      #10, #71, #80,
                 #28, #2e, #00,      #07, #c0, #00,      #00, #00, #00) ;

var f:text;                      esc,fs:char;
    lst,lstl:lstring (20) ;     i:integer;

begin
    esc:=chr (27) ;              fs:=chr (28) ;
    assign (f,'lpt1:');          rewrite (f) ; write (f,fs,'@') ;
    lst:='4rSlWTTl:;WV';        lstl:='7:00AM';

    write (f,fs,'&',fs,'2') ;
    write (f,chr (#0f8),chr (#0a1) ) ;
    for i:=1 to 24*3 do write (f,chr (ar [i]) ) ;

    writeln (f,lst) ;
    writeln (f,chr (#0f8),chr (#0a1),lstl) ;
    write (f,fs,chr (14) ) ;
    writeln (f,esc,'W1'chr (#0f8),chr (#0a1),lstl,eso,'W0') ;
    writeln (f, fs,'W1'chr (#0f8),chr (#0a1),lstl, fs,'W0') ;
    write (f,fs,'.') ;
    close (f)

```

end.

打印自造汉字



7:00AM



7 : 00AM



7 : 00AM

第五个程序，是用8针方式和24针方式，通过给出通用图形命令，按不同模式打印图形。这个程序中，每一图形的打印点的密度是不同的，打印效果一目了然，看的非常清楚。

程序中用到了两个过程，分别完成8针方式打印和24针方式打印，过程的参数是图形模式编号。

下面是该程序的源清单和它的运行结果。

```
program hanzi (input,output) ;
var f:text;          esc:char;
    lst:lstring (20) ; i:integer

    procedure draw8 (m:integer) ;
begin
    write (f,m:3,esc,'*',chr (m) ,chr (100) ,chr (0) ) ;
    for i:=1 to 50 do write (f, chr (170) ,chr (85) ) ;
    writeln (f)
end;

    procedure draw24 (m:integer) ;
begin
    write (f,m:3,esc,'*',chr (m) ,chr (100) ,chr (0) ) ;
    for i:=1 to 50*3 do write (f,chr (170) ,chr (85) ) ;
    writeln (f) ;
end;

begin
    esc:=chr (27)
    assign (f,'lpt1:') :rewrite (f) ;
    lst:='print graphic'; writeln (f, lst) ;

    writeln (f,' 8 dot modes' ) ;
    draw8 (0) ; draw8 (1) ; draw8 (2) ;
```


第八章 索引文件的实现及应用

在实现数据管理的操作中，最重要的任务，就是把大量的、貌似杂乱的、彼此无关的各种数据，组织成条目清晰、井然有序、结构性能很好的数据组织，以达到减少数据冗余、便于数据查找、追加、删除和修改等操作，以及有利于提高数据处理速度的目的。这涉及到数据库管理系统中众多的理论知识，超出了本书的讨论范围。我们在这一章中，只准备讲解文件的组织方式，即有关索引文件的某些有关内容。

索引文件，是计算机系统中常用的几种文件中比较复杂的一种，与顺序文件差异很大。让我们回过头来，回忆一番本书第四章讲解文件时提到过的问题。在那里我们曾说，MS PASCAL语言只支持顺序文件。又说，对文件可以顺序读写，也可以随机读写。还说到，随机读写的文件，也是顺序组织的文件，只是采用了按记录号进行读写的工作方式，同样不能在文件的中间位置插入或删除记录。而在实际应用中，把文件中的记录，按其内容组织为有序的排列方式是很需要的，我们却不能用顺序文件实现这种要求。引入与使用索引文件的目的，正是为了解决上述矛盾，使文件中的记录能按记录内容的大小关系依次排列，这必然涉及到把一个新的记录插入到文件当中某个位置，或把文件当中一个已有记录删掉等操作。在要读出某个记录时，我们就不仅能用顺序读的方式找出被读的记录，而且可以用指出记录内容的某个域的值的方式，以更快的速度找出要读的记录。此时，我们称这个域为记录的键，它的值被称为记录的键值。索引文件就是用键值对文件中的各个记录安排排放顺序的，并按键值读写相应记录。

8.1 索引文件的实现原理

索引文件实现记录插入与删除是怎么进行的呢？能否在插入一个新的记录时，把排在该新记录之后的全部内容都后移一个记录位置，以便为新记录“腾出”地方。在删除一个记录时，把被删记录之后的全部内容都向前“挤”一个记录位置，“挤掉”被删记录原来占用的空间呢？回答显然是否定的。原因有二。一是这样做太麻烦了，执行效率极低。文件可能很长，又保存在磁盘上，“移动”磁盘上的很多内容是很费时间的一种操作。记录的插入与删除，是使用索引文件过程中很频繁的动作，这种办法带来的系统低效率是不能忍受的。第二，这样组织起来的文件，要查找某一给定键值的记录，还是非常困难的一件事。按键值查找记录是使用索引文件各种操作的基础，必须以某种更好的办法处理。

解决问题的出路，是在数据文件本身之外，再给出一个索引表。索引表的作用，是指出每个键值对应的记录在数据文件中的位置。这很类似于我国的新华字典的组织方式。字典的内容，是给出收入的每一个汉字的读音和它的含义。为了能快速地从字典中查到每一个字，在字典开始，总要给出一个或几个索引表。例如，1957年出版，1979年12月修订第5版，以后几十次印刷再版的、供中等文化程度使用的《新华字典》，就是以汉语拼音字母音序进行排列的。它首先给出了一张“汉语拼音音节索引”表，索引表本身的内容安排是：

• 每一音节的内容，和读音为该音节的第一个汉字在字典正文中的页号。还同时给出了一个例字。例如：

a	啊	1
ka	咖	238
te	特	442

借用这张表，查找知道读音的字所表示的意思就快多了，不必翻找整本字典。

• 对音节本身在索引表中的排列次序，是按拼音字母从A到Z的次序排列的，使它和字典正文的内容的排序办法相同。我们也能不通过这张索引表直接方便地到字典正文中查找要找的字，即我们把字典翻到估计差不多的页号处，然后根据该页上的内容和我们要查的字的拼音接近的程度，再向前翻或向后翻若干页，用此办法翻不了几次，也能找到要找的字。

这张表不能用于查找不知道读音的汉字。字典中还必须给出按字形查字的另一张索引表。现在通用的有按“部首检字”和按“四角号码检字”两种方案。在部首检字方案中，检字索引表又被分成两部分，即首部目录和检字表，首部目录是检字表的索引，它给出检字表中使用的189部在检字表中的位置，检字表则给出收入的11100个左右的每个汉字在字典正文中的页号。通过检字索引表查找要找的汉字就快多了。

在汉字中，还有一部分字是比较难于说清它的部首的。说不清时，就无法用部首检字表查找这个字。为了解决这一矛盾，字典中还给出了另外一张索引表，“难检字笔画索引”，共270个字左右，如凸、世、来、串等字。这些字是按每个字的笔划多少来编排次序的。

讲索引文件之前，说了这么多新华字典的编排方法，是为了让读者充分理解“索引”二字所代表的准确含义，和索引表在实现快速查找中所起的重要作用。索引文件的概念和用法，使不少人感到迷惑，其实它和前边讲的新华字典的编排方法非常类似。区别也有一点，最重要的区别是，新华字典的全部内容已经由编著者编排好了，使用者能做的事只是查阅字典。而索引文件支持则是计算机系统提供的一种软件手段，不仅被用来查询有关索引文件中的内容，还被用来增删修改有关文件的内容，能使用户处在索引文件的创建者和使用者的双重地位。

索引文件，是在数据文件之外，又为它组织了一个索引表，索引表本身也是一个文件。因此，一个索引文件将由数据文件和索引表文件两部分内容组成。数据文件中，只按数据录入的实际次序，保存每一个数据记录，文件被组织成记录的一个序列。记录的次序仅取决于记录录入的顺序，与记录的内容无关。由此得出一个明显的结论，这个数据文件本身仍是顺序结构的文件。索引表文件的作用，是指出每个键值对应的记录在数据文件中的位置。索引表文件的内容，是按特定规则组织起来的。这些规则表现为：

• 索引表文件的每一个记录，都由一个键值域和一个指引字域组成。指引字域的值，用于指出对应于给定键值的记录在数据文件中的排列位置，或用于指出在索引表文件中的另外一部分内容的相对位置。

• 索引表文件中的记录，是严格按照键值域的值的大小依次排列的。这里有在文件当中位置真正的记录插入和删除等操作。由此得出结论，索引表文件不能是顺序文件。实现中要解决的首要矛盾，是要能方便、高效率地按键值对记录排序，按键值查找，和实现记录的插入和删除操作。

从使用索引文件的角度看，我们真正关心与使用的还是数据文件中的内容。索引表文件，不过是读写数据文件过程中用到的一个中间环节，使读写不是按键值对记录排序的数

据文件的操作，能够按记录键值大小的次序执行。简言之，使用索引文件的目的是，是实现按记录键值执行数据文件的读写操作。它不同于对顺序文件的顺序读写，也不同于对随机文件的直接读写（对顺序文件按记录号读写）。也就是说，对索引文件，是按记录内容确定读写哪一个具体的记录，而不再是直接按记录在数据文件中的排列位置来确定了。

说到底，索引文件，就是为顺序的数据文件再给出一个附加的索引表文件。在读写这个数据文件时，必须经过索引表文件，首先按记录键值确定这一个键值在索引表文件中的准确位置，找出相应记录号，然后才在数据文件上执行真正的读写操作。不经过索引表文件就直接读写数据文件是没有道理的。支持索引文件的核心问题，就是给出建立和使用索引表文件的必要能力。

实现与管理索引表文件的方案有多种，用得最普遍的是采用 B^+ 树的方案。

B^+ 树是对普通 B 树的一种发展。使用 B^+ 树实现索引表文件的最大好处是树的对称性，（平衡性），即索引树总保持一个等腰三角形的形状，所有的叶结点都处在同一层中。

B^+ 树是由多个结点组成的，每个结点又由若干个索引项组成，每个索引项都是由一个键值字段和指引字（指针）字段组成。下面是一棵 B^+ 的示意图。



下面给出讨论 B^+ 树用到的几个概念。

树最顶上的一个结点，是根结点，中间的被称为中间结点，最底层的是叶子结点。有时把根结点和中间结点又统称为非叶子结点。

把一个结点内最多能包含的索引项的数目，称为该树的秩（ORDER）。请注意，在有些书上可能采用其它办法定义秩。

一棵秩为 d 的 B^+ 树具有如下特征：

- （1）每个结点中最多包含 d 个索引项；
- （2）除了根结点之外，每个结点中至少包含 $d/2$ 个索引项，根结点至少包含一个索引项。根结点中尚无索引项时，为空的 B^+ 树；
- （3）含有 i 个索引项的非叶子结点有 $i+1$ 个儿子结点；
- （4）所有的叶子结点都处在同一层上，它们都没有自己的儿子结点。

从使用 B^+ 树的观点看，非叶子结点中每个索引项的指引字，用于指向树的一个下层结点。叶子结点中的索引项的指引字，用于指向数据文件中的一个数据记录。此外，还可以用指针把每个叶子结点勾链起来，使其成为双向链表的结构，以便实现文件的顺序检索、读出操作。此时，要记下叶子首结点和尾结点位置。

下面介绍在 B^+ 树上进行的操作。并假定每个结点内的索引项，是按键值字段的值以升

序排列的，如下图所示：



图中 P_1, P_2, \dots, P_{j+1} 代表该结点内的 $j+1$ 个指引字。 K_1, K_2, \dots, K_j 代表 j 个键值。对非叶子结点，若结点内有 j 个键值，它要用 $j+1$ 个指引字，对叶子结点， j 个键值将只用 j 个指引字。按前面说的，此处将有 $K_1 < K_2 < \dots < K_j$ 的关系。

一、查找操作

要查找键值为 K_x 的记录，就要从根结点查起，要与结点内的每个键值比较（当树的秩较大时，可选用折半查找法，否则可选用顺序查找法）其结果可能为：

(1) 若 $K_x \leq K_i$ ($i = 1, 2, \dots, j$)，结点又不是叶子结点，则沿指引字 P_i 向下一个结点继续查找。若已找到叶子结点， $K_x = K_i$ ， P_i 是键值为 K_x 的记录在数据文件中的位置信息（如记录编号）。否则找不到。

(2) 若 $K_x > K_i$ (i 不等于 j)，按 i 增加的方向继续在结点内向右查找。若 $K_x > K_j$ ，结点又不是叶子结点，则按指引字 P_{j+1} 继续向下一层结点查找。若为叶子结点，表示是查不到。

二、插入操作

要插入键值为 K_x 的索引项到 B⁺ 树中，先执行查找，若找到了，又规定键值不得有重复情况，则表明不能插入，指明出错。否则，记下最后查找的叶子结点。若结点内索引项未滿，则按 K_x 的值把它插在该结点适当位置。要保证结点内键值严格的排序关系，即把键值比 K_x 大的索引项后移一个位置，以便插入 K_x 索引项。若结点内索引项数已滿，先要执行分裂 (SPLIT)，即申请一个新结点，把原结点中一半数目的索引项分到新结点中去，再把 K_x 索引项插入到原结点或新结点中（取决于 K_x 的值和两个结点分裂处的键值的大小关系）适当位置。此外，还要在上层结点中插入一个新的索引项，修改那里的指引字内容等。这是一个键索反应，要用递归算法解决，有可能要生成一个新的根结点，使树的高度长高一层。这在后面给出的程序例子中可以看得很清楚。

三、删除操作

要删除键值为 K_x 的索引项，首先执行查找操作。若查不到，则是删除错误。若在叶结点中查到了，则执行删除操作。实际上就是把结点内键值比 K_x 大的所有索引项都左移一个位置。但记住，此时删除操作不算完成，还要检查执行本次删除后，剩在结点内的索引项数目是否还等于或大于树的秩的值的一半，若大于或等于，则结束本次删除操作。不足一半时，或从左或从右部结点中调剂一或几个索引项过来，此时可能要修改上层结点中的有

关键值；或取消本结点，把剩下的索引项送给左或右邻结点。此时将引起在上层结点中删除一个索引项，这可能是又一个链索反应，要用递归算法处理，最终还可能导致取消原来的根结点，使树的高度降低一层。这些也可以在后面给出的程序例子中看清楚。

B⁺树中的插入和删除，总能保证树的平衡，它的管理与实现对计算机硬件有较大的独立性。对B⁺树内容的变更与维护都比较简单，运行效率高。所以B⁺树在实现索引表文件中占有重要的地位。

对B⁺树维护、操作的效率，与两个参数关系密切。一个是树中总共要给出的最大的索引项的数目N，另一个是树的秩d。二者的关系可粗略地表示成：

$$N \leq [d/2]^{h-1}$$

这里的h表示的是树的高度。它与查找B⁺树时读写结点的次数成正比，应选得小些。为了保证所要求的N值，d必然要选得大些，这又会使得在同一个结点内的查找、插入与删除中索引项的左右“移动”的时间变长。而且，一个结点选多大，还应与磁盘的物理入/出块的大小配合好。磁盘经常是以块（BLOCK）为物理读写单位，一个结点用一块还是几块盘空间，还是一块内分成几个结点等等都是要考虑的实际问题。假定N选为128K，树的秩选为64，树的高度是几呢？

从 $128 \times 2^{10} \leq (64/2)^{h-1}$ 得h最小为5。

四、顺序查找

在进行顺序查找时，若每次都从根结点查起，效率是很低的。为此，我们可以用指针把所有的叶子结点勾链起来，建成双向链表的结构，并记下叶子首结点和叶子尾结点，我们就能方便地实现按键值升序关系或降序关系访问整棵树的叶子结点，实际上就能按键值的升序或降序关系完整地读出数据文件中的全部记录。

8.2 支持索引表文件操作的程序

在前一节已反复说到，使用索引文件的关键问题，是实现在索引表文件上的检索、插入和删除三项最基本的操作。必须用某种方式向用户提供执行这些操作的能力。MS PASCAL语言没有直接提供这类功能。我们在本节将向读者介绍实现这类功能的具体方法。将给出一个很好的程序，既用来讲清楚实现索引文件支持的原理，又为读者提供一个切实可行的程序雏形。事实上，只要再解决一点实用中的小问题，这个程序中的绝大部分内容是真正可用的。

突出的矛盾是这个程序比较长。我们已尽了很大的努力，使程序尽量变短，使程序实现的功能尽量简单，但最后设计出的程序还是偏长。读者要学好这一节的内容，可能要花费比较多的时间。这一节中给出的内容是很具体、实用的，在其它书籍资料中很难找到。

下面先给出这个程序的清单。

```
program bptree(input,output);
const order=8; limite=order div 2;
      call=5;
type item=record n,p:integer end;
```

```

node=record
    ar:array [1..order] of item;
    pp:integer ;
    llnk,rlnk,layer,nb: integer
end;
treedsc=record
    root,tail,nextblk,nextdp: integer
end;
var
    tree: treedsc ;
    i,l,lay,bk,xx,xx1: integer;
    idx : array [1..80] of node;
    path : array [0..tail] of
        record
            blk,loc,dp : integer
        end;
    splitdo : boolean;
    procedure search(kx,blk : integer);
var i : integer;
begin
    with idx[blk] do
        [ i := 1;
        while (kx>ar[i].n)and(i <=nb) do i := i + 1;
        path[layer].blk :=blk;
        path[layer].loc := i ;
        if (i <=nb) and (kx=ar[i].n)
        then path[layer].dp :=ar[i].p
        else path[layer].dp := 0 ;
        writeln(' search-----> blk,loc : ', blk, loc);
        if layer < > 0
        then search(kx,ar[i].p)
        ]
    end;
    procedure inzert (kx : integer) ;
var i, k, oldblk, newblk, jj : integer;
    genroot : boolean ;
    procedure split (lay : integer) ;
var m : integer ;
begin

```

(* 执行结点分裂 *)

```

newblk := tree.nextblk ;
tree.nextblk := tree.nextblk + 1 ;
for m := 1 to limite do
    idx[newblk].ar[m] := idx[oldblk].ar[m + limite] ;
idx[oldblk].nb := limite ;
idx[newblk].nb := limite ;
idx[newblk].layer := idx[oldblk].layer ;

    (* 如为叶结点, 修改链指针 *)
if idx[oldblk].layer = 0
then [ idx[newblk].llnk := oldblk ;
      idx[newblk].rlnk := idx[oldblk].rlnk ;
      idx[oldblk].rlnk := newblk ;
      if oldblk = tree.tail
      then tree.tail := newblk
      else idx[idx[oldblk].rlnk].llnk := newblk ] ;
! writeln('lay, oldblk, newblk ', lay, oldblk,
          newblk)
end ;

begin      (* insert procedurs body *)
    genroot := false ;
    if not splitdo
    then [ search(kx, tree.root) ;      lay := 0 ] ;
    if not splitdo and (path[0].dp < > 0)
    then writeln('? This number exists, no inserted')
    else
        [ oldblk := path[lay].blk ;      i := path[lay].loc
          ! writeln('insert-----> ',
                    lay, path[lay].blk, path[lay].loc) ;
          if idx[oldblk].nb = order
          then [ genroot := oldblk = tree.root ;
                  split(lay) ; (* 转去执行结点分裂 *)
                  splitdo := true ]
          else splitdo := false ;

              (* 重新确定KX插入的结点和位置 *)
          if splitdo and (kx > idx[oldblk].ar[limite].n)
          then [ jj := newblk ;      i := i + limite ]
          else jj := oldblk ;

              (* 执行索引项的插入 *)

```

```

        writeln('blk= ', 2, '    loc= ', i:2);
with   idx[jj] do
  [ if nb < > 0 then
    for k :=nb downto i do ar[k+1] :=ar[k] ;
    ar[i].n :=kx ;          nb :=nb+1;
    if lay=0
      then [ ar[i].p := tree.nextdp ;
              tree.nextdp := tree.nextdp+1 ]
      else ar[i].p :=bk
    ] ,

        (* 生成新的根结点并写入相应的值 *)
if genroot then
  with idx [ tree.nextblk ] do
    [ llnk:=0; rlnk:=0;    nb:=2;
      ar[1].n:=idx[oldblk].ar[limite].n ;
      ar[1].p:=oldblk ;
      ar[2].n:=maxint ;
      ar[2].p:=newblk ;
      layer :=lay+1 ;

      tree.root :=tree.nextblk ;
      tree.nextblk :=tree.nextblk+1 ;
      genroot:=false ;    splitdo:=false
    ] ,

        (* 如果有过分裂操作, 要在上层结点执行插入 *)
        (* 还可能要修改上层结点中的有关的内容 *)
if splitdo then
  [ lay :=lay+1 ;
    jj :=path[lay].blk ;    i:=path[lay].loc ;
    if newblk < > tree.tail
      then [ idx[jj].ar[i].n
              :=idx[oldblk].ar[limite].n ;
              path[lay].loc :=path[lay].loc+1 ;
              bk :=newblk;
              kx :=idx[bk].ar[idx[bk].nb].n ]
      else [ idx[jj].ar[i].p :=newblk ;
              bk :=oldblk ;
              kx :=idx[bk].ar[idx[bk].nb].n ] ,
    inzer(kx)
  ]
  (* 递归调用 *)

```

```

    ]
end ;

procedure delite(kx : integer) ;
var i, j, k:integer ;
    tailnode : boolean ;

    procedure merge(lay:integer) ;
var m, m1, m2, n, n1, n2, j, k : integer ;
begin
    j:=path[lay].blk;
n:=path[lay+1].blk ; m:=path[lay+1].loc ;
with idx [n] do
    [if m< >nb then m2:=m+1 else m2:=m-1,
    n1:=ar[m2].p];
    1 writeln(,merge----->n, m, n1:', n, m, n1);
if n1<j then n2:=j else[n2:=n1 ; n1 :=j ];
tailnode :=idx[ n ].ar[idx[ n ].nb].p=n2;
    1 writeln('tailnode----->', tailnode);
    1 writeln('j, n1.n2 ', j, n1, n2);
    1 writeln('idx[n1].nb, idx[n2].nb',
    1 idx[n1].nb, idx[n2].nb);
if idx[n1].nb+idx[n2].nb<order
then [ (* 执行合并 *)
    i :=idx[n1].nb,
    for k:=1 to idx[n2].nb
    do idx [n1] .ar[i+k] :=idx[n2].ar[k];
    idx[n1].nb :=i+idx[n2].nb;
    if m>m2 then m1 :=m else m1 :=m2;
    if tailnode
    then
        idx[ n ].ar[m1-1].n:=idx[ n ].ar[m1].n
    else
        idx[ n ].ar[m1-1].n:=idx[n1].ar[idx[n1].nb].n
        idx[ n ].ar[m1-1].n:=idx[n1].ar[idx[n1].nb].n ;
        1 writeln('m,m2, m1----->', m, m2, m1);
        1 write('idx [ n ] .ar [m1-1] .n-----') ;
        1 writeln(idx [ n ] .ar [m1-1] .n)
        (* 为叶结点则修改链指针 *)

```

```

if   idx[j].layer=0 then
    [   idx[n1].rlnk:=idx[n2].rlnk;
      if idx[n1].rlnk< >0
        then idx[idx[n1].rlnk].llnk:=n1
        else tree.tail:=n1
    ],

    (* 继续在上层结点执行删除 *)
with idx[n] do
    [nb:=nb-1;
      if m1<=nb
        then for k:=m1 to nb do ar[k]:=ar[k+1];

      if n=tree.root
        then
          [ if nb=1 then tree.root:=ar[1].p]
          else
            if nb < limite
              then merge(lay+1) (* 递归调用 *)
          ]
    ]
else [ (* 均衡两个结点的内容 *)
      m :=path[lay+1].loc;
      if j=n1 then
        [with idx[n1] do
          [ nb:=nb+1 ;
            ar[nb]:=idx[n2].ar[1];
            idx[n].ar[m].n:=ar[nb].n;
            |   writeln(' 均衡 idx[n].ar[m].n',
              |           n,m,idx[n].ar[m].n)
          ],
          with idx[n2] do
            [ nb:=nb-1;
              for k:=1 to nb do ar[k]:=ar[k+1] ]
        ]
      ] else
    [with idx[n2]do
      [ nb:=nb+1;
        for k:=nb down to 2 do ar[k]:=ar[k-1];
        ar[1]:=idx[n1].ar[idx[n1].nb];
        |   writeln(' 均衡 idx[n].ar[m].n',

```



```

                                , n, m, idx[n].ar[m].n)
    ],
    with idx[n1] do
        [ nd:=nb-1 ,
          idx[n].ar[m-1].n:=ar[nb].n ]
    ]
]
end,
begin      (*delite procedure body*)
    tailnode :=false ;
    search(kx, tree.root); lay:=0;
    if path[lay].dp=0
        then
            writeln('? This number no exists, no deleted')
        else
            [ ! writeln('delete----->', kx);
              ! writeln ;
              j:=path[lay].blk ;
              i:=path[lay].loc ;
              with idx[j]do
                  [ nb:=nb-1 ;
                    if i<=nb then
                        for k:=1 to nb do ar[k]:=ar[k+1] ;

                    if j<>tree.root
                        then
                            if nb<limite
                                then merge(lay)
                                else [if (i=nb+1) and (j<>tree.tail)
                                      then idx[path[lay+1].blk].
                                          ar[path[lay+1].loc].n:=ar[nb].n ]
                                      else if nb=1 then tree.root:=ar[1].p ;
                                ]
                            ]
            ]
        end,

    procedure trace(blk:integer);
    var i: integer;
    begin
        with idx[blk] do

```

```

[write('blk=', blk:3),
for i:=1 to nb do
  [write(ar[i].p:8, ar[i].n:6),
  if (nb>4) and (i=4) then writeln],
  writeln,
for i:=1 to nb do
  if layer( )>0 then trace(ar[i].p)
  ]
end ,

procedure acendant ,
begin
  writeln ,
  l:=1,
  writeln('-----acendant order output-----'),
  while l( )>0 do
    with idx[l]do
      [ write('blk=', l:3),
      for i:=1 to nb do write(ar[i].n:8),
      writeln, l:=rlnk]
    end ,

    procedure decendant ,
    begin
      writeln ,
      l:=tree.tail,
      writeln('-----decendant order output-----'),
      while l( )>0 do
        with idx[l] do
          [ write('blk=', l:3),
          for i:=nb downto 1 do write(ar[i].n:8),
          writeln, l:=llnk ]
        end,
      BEGIN (*main program*)

        (*part 1: initialize the variables *)
        with tree do[ root:=1, nextblk:=2,
          tail:=1, nextdp:=1],
        splitdo:=false ,
        with idx[1]do
          [nb:=0, llnk:=0, rlnk:=0, layer:=0] ,
          (*part 2:create the index-tree*)

```

```

write('xx1= ') ; readln(xx1) ;
for xx:=1 to xx1 do inzert(xx) ;
trace(tree.root) ;

(* part 3 : delete some items in the insex-tree *)
writeln('开始删除操作') ; xx :=xx1 ;
while xx>0 do
    [delite(xx) ;
     xx :=xx-2] ;
trace(tree.root) ;

(* part 4 : display the contents of all leaf nodes *)
write('按升序显示全部叶子结点的内容') ;
acendant ;

write('按降序显示全部叶子结点的内容') ;
decendant ;

writeln('把xx1以内的正的偶数重新插入到树中')
xx := 2 ;
while xx<=xx1 do [inzert(xx) ; xx :=xx+2] ;
trace (tree.root) ;
acendant
end.

```

为了简便和容易看清B+树中的内容，我们在程序中做了如下规定：

- 树的秩ORDER为8。
- 树的最大高度为6 (tall+1)。
- 树IDX最多可以有80个结点，这些结点是以数组形式提供的。
- 每个节点NODE由ORDER个索引项，一个指引字pp，和左指针、右指针、结点层号和结点内已有索引项数目四个结点描述信息组成。
- 每个索引项都由键值n和指引字p两个域组成，并把它们都定义成整数类型。
- 整个索引树TREE，用根结点号、最右（尾）叶子结点号、下一个可用的结点号和下一个可用的数据文件的记录号四项信息描述。叶子首结点号总为1。
- 查找过程中，从根结点到叶子结点所走过的路径记在PATH中，登记的是结点号BLK、结点内的索引项位置LOC和所查到的指引字DP三项内容。查不到时，DP取零值。

有了这些说明，读者就能方便地看清在程序中说明部分给出的每个标识符代表的含义。

在程序中，给出了六个过程。其中SEARCH、INZERT和DELITE的功能，是分别完成查找、插入和删除操作的三个过程，是我们程序中的核心部分。过程名用INZERT和DELITE取代INSERT和DELETE，是因为INSERT和DELETE是MS PASCAL中用来实现字符串操作的标准标识符。这两个过程，又各自说明了自己的一个内部过程SPLIT和MERGE，它们分别完成插入过程中出现的结点分裂，和删除过程中出现的结点

合并或内容均衡的操作过程。另外三个过程TRACE、ACENDANT和DECENDANT，是用来显示索引树中已有内容的辅助性过程。TRACE过程可以显示整棵树每一个结点的内容，ACENDANT和DECENDANT过程可以按键值升序或降序的顺序，显示树中全部叶子结点的内容。

下面详细介绍这些过程的实现算法。

查找过程SEARCH

查找，是在应用索引文件中经常用到的操作，它又是进行插入和删除操作的必经步骤。

这个过程用到两个参数，一个是要查找的键值KX，另一个是开始本次查找的起始结点号BLK。查找总是分成两个步骤进行：

(1) 在指定的结点内，按给出的键值找到一个索引项位置。在我们的程序中，是通过一个WHILE语句顺序进行的。一个节点内的各键值是严格按升序排列的，这样，找到的索引项可能是：

- 索引项中的键值等于要查找的键值；
- 索引项的键值，是比要查找的键值大的全部键值中最小的那一个，这是在该结点内查不到的体现。为了程序更简单，我们省略了另外一种可能，即检索出的索引项，是在结点内最大键值之后的位置上。这种省略并不影响本程序的执行功能。实际上，这是取消了B⁺树非叶子结点中的第j+1个儿子结点（假定它里边已有j个索引项），使叶子结点和非叶子结点的分裂与合并操作，能够用相同的办法处理，程序简单一点，可以不用全部结点中的pp指引字。

(2) 在结点间，按有关索引项中的指引字的值实现递归查找。B⁺树中保存的每个键值，至少在叶子结点中有一记载，在非叶子结点中可能有、也可以没有记载。因此，查找应从根结点开始，不管在非叶子结点中找到或没找到要查找的键值，都应按相应指引字的值执行结点间的递归查找，查找完某一叶子结点才结束这个递归过程。结点间的这种递归查找，是用这个过程中的最后一个语句完成的。

在这两个查找步骤中，要查找的键值KX是必须给出的。为了实现结点间的递归查找，每次都应该给出本次查找的起始结点号，BLK参数也是必须有的。

在实现一次完整地查找过程中，总是从根结点查起，接下来是用递归的算法查找下层一个结点。查找所经过的路径和查找每个结点得到的结果，都记录在PATH数组变量中。数组的下标值用的是结点的层号LAYER的值。要记下本次所查的结点的结点号BLK，记下结束该结点内查找的索引项的编号I，和是否查到了的标记值DP，查不到时DP赋零值，查到时DP为一个指引字的值（对非叶子结点，为下一层一个结点的结点号，对叶子结点，为一个数据记录在数据文件中的位置信息，例如记录编号）。在我们的程序中，只有对叶子结点得到的DP值有用，对非叶子结点得到的DP值未做任何处理，也没有用到它。查找过程体内当中的五行语句，用于完成登录PATH数组的有关内容。记下查找路径，对简化结点分裂和结点合并、结点内容均衡是十分重要的。

对查找过程最后要说明的两点是：第一，当树的秩较大时，结点内查找以采用折半查找为好，这可以改善查找的速度。粗略地算，若树的秩为128，顺序查找平均操作次数为128的一半，即64次，而用二分法，则为 $\log_2 128$ ，即7次。第二，树的每个结点由ORDER个索引项和一个指引字，以及另外四个结点描述信息组成。在ORDER个索引项之外

再多给一个指引字，是因为B*中的每个非叶子结点内，如果有*i*个索引项，它要有*i* + 1个指引字，当*i* = ORDER时，多出的那个指引字正好用pp表示。前面已提到，我们没有使用和处理pp，程序变得简单一点，带来的问题是索引表文件变得大一点，每个索引项满了的非叶子结点浪费了一个指引字。在我们的程序中，取消每个结点中的pp是完全可以的。

真正理解这个查找过程的实现算法，对看懂这个完整的程序是大有裨益的。

插入过程INZERT

插入过程的功能，是把给出的一个键值插入到索引树的适当位置。从使用索引文件的角度看，在把键值插入到索引树中以后，还应该取得一个在数据文件中的数据记录位置的有关信息，再把这个键值对应的记录写入到数据文件中。在我们的程序中，未给出对数据文件的写入操作，以便把我们的精力更多地集中到对索引树本身的操作上。先看执行一次插入操作的具体步骤：

实现插入之前，首先要在索引树中进行查找，检查要插入的键值是否已经存在了。如果已经有了，表明出现键值重值情况，我们在程序中给出了一个出错信息，并且不执行这次插入。只在查找不到的情况下才执行本次插入操作。

插入总是从叶子结点开始的。前边的查找过程，已把查找走过的路径和查找的结果登记在PATH数组变量中。我们可以从PATH[0]元素中取得本次插入用到的叶子结点的结点号，以及在该结点内插入的准确位置。当该结点内已有的索引项数目小于树的秩时，就可以把要插入的键值和一个指向数据记录的指引字的值（怎样得到这个值到后面再讲）插入到该结点内。例如，要把键值15插入到如下一个结点中：

						NB	
3	8	12	20	39			5

就要把键值比15大的两个索引项后移一个位置，把15和相应指引字插入到原来20和相应指引字占用的位置上，并把该结点内已有索引项数目NB增1。此时结点内容变为：

						NB	
3	8	12	15	20	39		6

当结点内尚无任何一个索引项时（此现象只出现在向空索引树插入第一个索引项的情况下），当然应把要插入的内容写在结点内头一个索引项的位置。这与结点内已有若干个索引项的情况下的插入操作的差别主要有两点：

- 插入的位置必定为第一个索引项；
- 没有把某些索引项后移一个位置的动作，这在程序清单中可看得很清楚。

当要插入的结点中索引项数目已满，则不能用上述办法处理。此时，应首先执行结点分裂，就是把该结点内一半数目的索引项分到另外一个新的结点中去，这还涉及到修改上层结点、甚至生成一个新的根结点的操作。分裂操作是由SPLIT过程完成的，我们将在介绍完插入操作之后，再详细讲解分裂算法。分裂过程结束之后，就可以执行所要的插入操作，这时的插入与前边讲的插入操作是类似的。差别也是有的。主要表现在：

- （1）在不分裂而直接插入的情况下，结点号和插入位置已经确定。而在分裂之后也

插入，有可能要变化插入的结点和插入的位置，例如，要把36插入到如下一个结点中：



此时，结点号应为该结点的编号，位置应为6。分裂后，原结点中只剩下3、8、12和15四个索引项，20、39、45和60四个索引项分到另外一个新结点中。这时36应插入到这个新结点中的第2个位置上。假如插入的不是36而是10，则插入的结点号和位置不会因结点分裂而变化。为此，区分是直接插入还是分裂后的插入是必要的，这是通过布尔变量SPLITDO完成的。在分裂后的插入过程开始之前，先要重新确定K，插入的结点和位置。

(2) 插入之后的后续处理不同。在直接插入的情况下，插入完成之后没有其它后续处理。在经过分裂之后的插入操作中，本层插入完成之后，还必须在上层结点中继续执行新的插入。因为分裂时，在本层中多出了一个新的结点，必须把它的有关信息登录到上层结点中去。这又可以按本次分裂是出现在根结点中，还是出现在非根结点中执行两种不同的后续处理。如果是在根结点中出现分裂，它还没有上层结点，应执行生成新的根结点并写入相应的值的操作，否则，要在上层结点中执行新的插入，还要修改上层结点中的有关内容。这是通过递归调用INZERT过程完成的。这个递归调用过程，将在遇到某一层中不再出现结点分裂，或者生成了一个新的结点的情况下才结束。

在我们的程序中处理到的具体细节问题，有如下几个：

- 在调用分裂过程SPLIT时，要置变量SPLITDO为真，并按分裂是否发生而在根结点中设置GENROOT变量的值，为返回之后的操作提供依据。

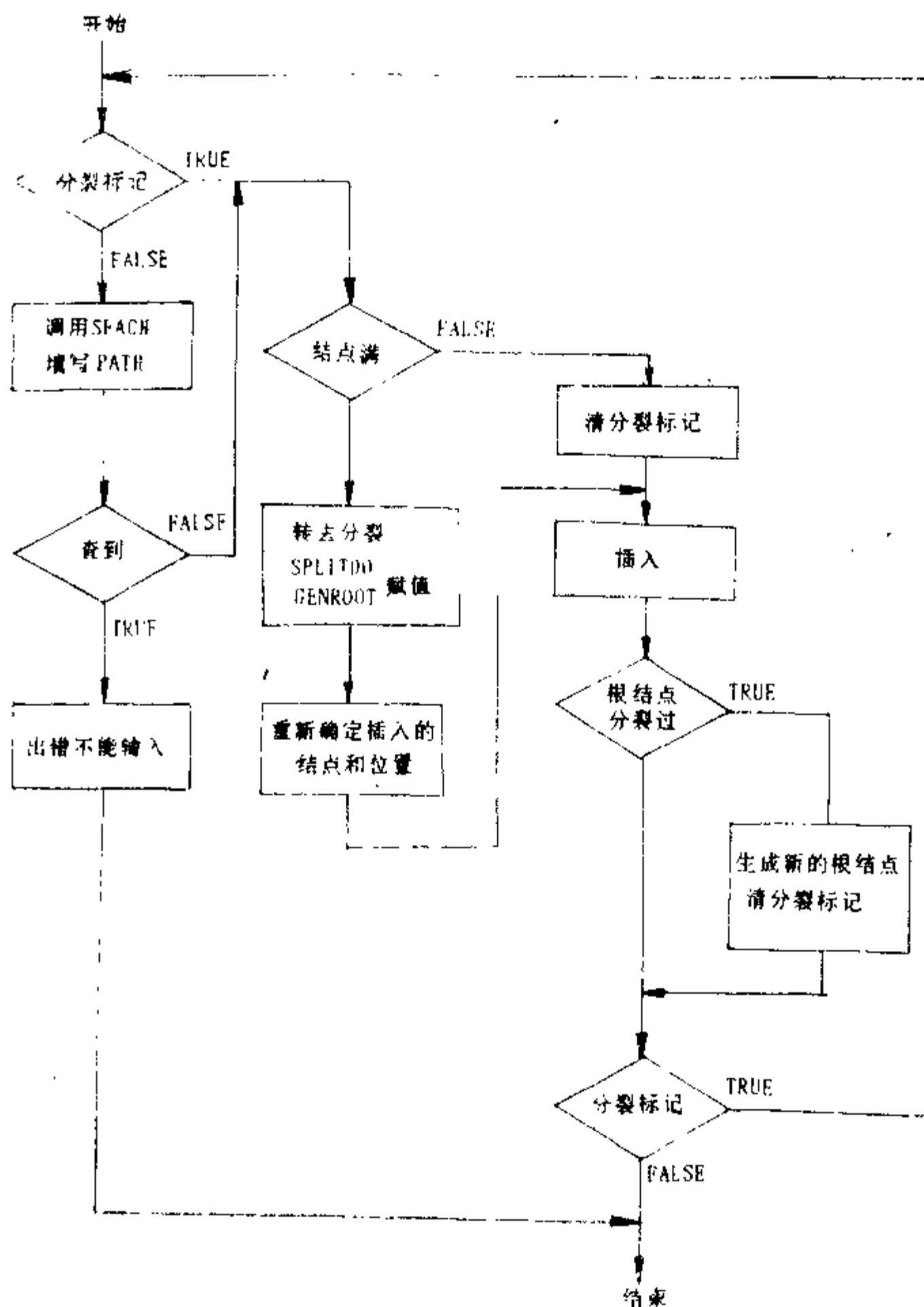
- 在生成新的根结点时，置已有索引项的数目为2，置第二个索引项的键值为MAXINT。目的是使从根结点开始的全部查找，不会出现查到的索引项在结点内已有数据项之后的局面。这就可以避免使用结点内的pp指引字，使每个非叶子结点拥有的儿子结点数目等于它已有的索引项数目，程序就可以稍为简单一点。这样处理不会损害程序的正确性。MAXINT是否是该索引树的一个真正的键字，将只取决于叶子尾结点中是否有此值。

- INZERT过程既用来执行叶子结点中的插入，也用来执行非叶子结点中的、分裂带来的插入。这两种插入开始的背景是不相同的。插入总是由叶子结点内的插入开始，插入之前必须首先经过查找过程，以检查是否有键值重值错误。在无重值即允许插入的情况下，也要查到插入的叶子结点号和插入的位置，并记下查找走过的路径。而在分裂造成的插入过程中，上述信息都可以在PATH变量中找到，不需要调用查找过程。我们在程序中，是用SPLITDO变量来区分这两种情况的，并按它的值，决定是否要首先调用SEARCH并检查键值是否重值。

- 如果执行的分裂发生在根结点上，则要生成新的根结点。此时一定要把新的根结点编号写入TREE.ROOT变量中。在写完新的根结点的内容之后，还应把GENROOT和SPLITDO变量变为假，以保证不再调用INZERT过程。

- 如果分裂是发生在最右的叶子结点中，我们将把原结点的相关内容作为向上层结点插入的内容，否则，将把新结点的相关内容插入到上层结点中。这样做的目的仍是为了避免使用pp指引字。

下面给出INZERT过程的程序流程框图。



分裂过程SPLIT

分裂过程的主要功能，是把一个索引项数目已满的结点中一半数目的索引项，划分到新分配的一个结点中去。

分配一个新结点的办法，就是把下一个可用结点用作新结点，程序中是用NEWBLK:=TREE.NEXTBLK语句完成的。这之后，还要使TREE.NEXTBLK的值增1。分裂时，我们是把原结点中键值大的那一半数目的索引项，移到新结点的前半部分中，并修改原、新两个结点中实有索引项数目为LIMITE，还要使新结点的层号等于原结点的层号。

如分裂的结点是叶子结点，我们还在分裂过程中处理了叶子结点间的勾链关系。若原

结点为叶子尾结点，我们又修改叶子尾结点为分裂出来的新结点。

至此，我们可以说明对索引树本身的四项描述信息的用法了。

TREE.ROOT给出的是索引树的根结点编号，在每次生成新的根结点时，要修改它的值，查找总是从根结点开始。

TREE.TAIL给出的是索引树最右端的叶子结点编号，每当对最右端的叶子结点执行分裂操作时，要修改它的值。这个结点号，在按降序关系查找树中的全部键值时要用到它。

TREE.NEXTBLK给出的是在申请新结点时，下一个可用的结点编号，每申请完一个新结点，要对它执行加1操作。我们是按顺序来分配每一个新的结点的。

TREE.NEXTDP给出的是在数据文件中，下一次可写入一个数据记录的位置信息。我们将在后面部分再详细说明它的用法。

顺便说明，索引树的第一个结点，总是在最左端的叶子结点。我们用到的分裂算法，和后面将看到的结点合并算法，都能保证上述用法的正确性。

从分裂算法可以清楚地看到，一棵索引树，总是从叶子结点分裂开始，并逐层向上分裂而形成的，即索引树是“倒着”向上生长的。

删除过程DELITE

删除过程的功能，是把索引树中那个键值与给定键值相等的索引项删除掉。一般来说，这不会牵扯到对数据文件本身的操作，索引项被删掉之后，数据文件中相应的数据记录已经变得不能访问了。

删除操作是插入操作的逆过程。在实现删除之前，首先要在索引树中进行查找。若找不到给定的键值，表明遇到无此键值的一个错误，也就不能执行删除操作。查到了，则应执行删除操作，即把结点内比给定键值大的全部索引项左移一个位置，并把结点内的索引项数目减1。此后应检查结点内剩下来的索引项数目，当它小子秩的 $1/2$ 且结点不为根结点时，应把它与左右邻结点合并，或从那里“补充”过来一个索引项，保证一个结点内索引项数目不得小于秩的 $1/2$ 的要求，这是通过MERGE过程完成的。若剩下来的索引项数目大于或等于秩的 $1/2$ 时，还要检查被删索引项是否是该结点内最右的一个索引项，如果是，还要修改上层结点中相应索引项的键值。在执行删除操作的结点为根结点时，只要结点内索引项数目不为1，就可以直接结束本次删除操作。若原根结点中检索项数目已变为1，它又还有下层结点，则应使它的下层结点（此时只能有一个）成为新的根结点。

这个过程不是很难看懂，我们不给出它的程序流程框图。这个程序段中的一个细节问题是，当被删除的结点为树的叶子尾结点时，不能修改它的上层结点中相应索引项中的键值。目的仍然是为了避免使用pp指引字。

结点合并或均衡过程MERGE

这是删除过程的一个内部过程。在删除过程删除了结点内一个索引项，出现剩下来的索引项的数目小子秩的 $1/2$ 时，要调用它来完成结点合并或内容均衡的操作。它比实现结点分裂的过程要复杂一些。

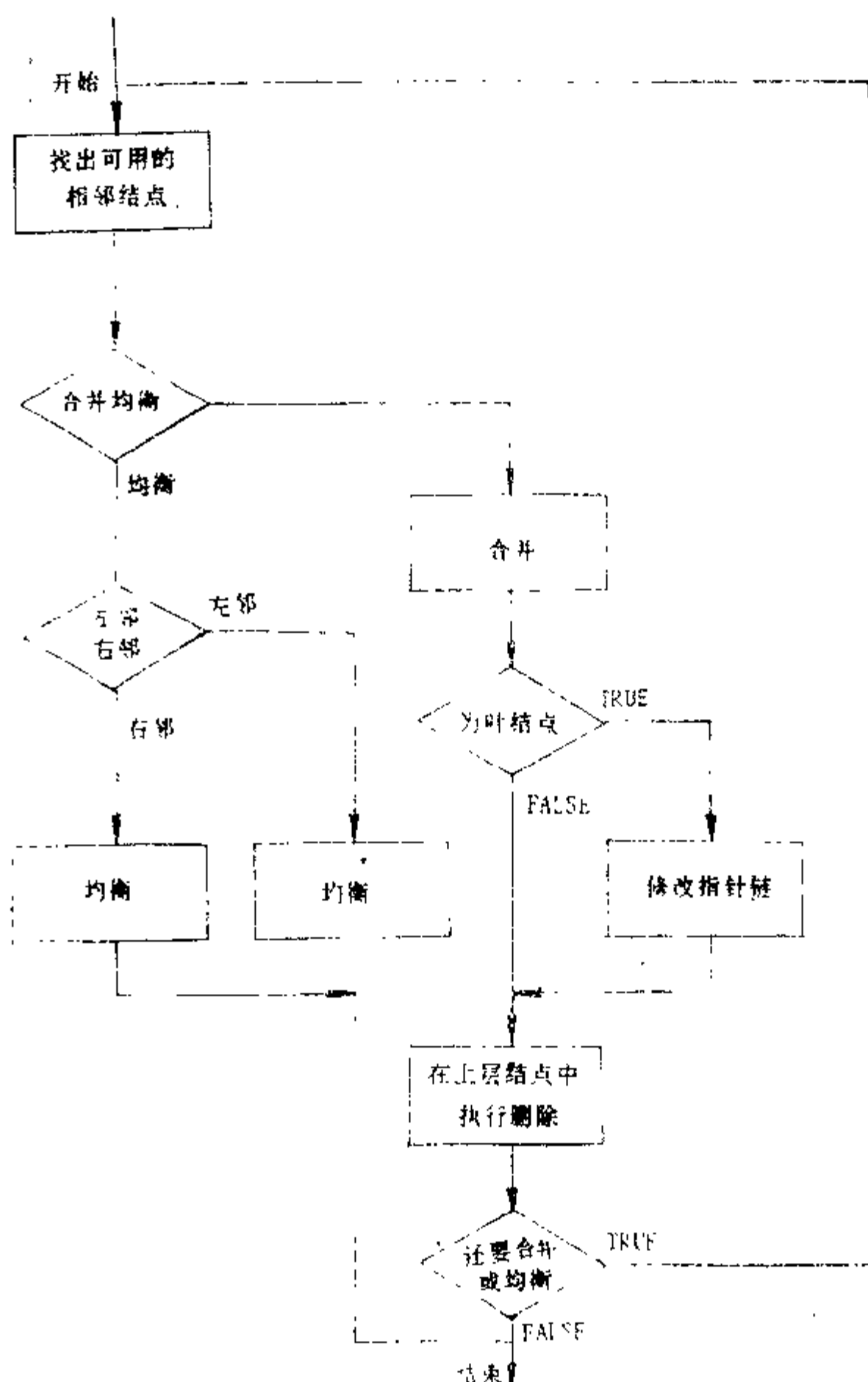
执行结点合并或均衡之前，首先要找出一个相邻的结点。它可能是原结点的左邻或右邻结点。我们采用的算法是，在存在右邻结点时总用右邻结点，只是在不存在右邻结点时，我们才选用左邻结点。找相邻结点，要用到上层结点中的信息。过程开始的几个语句，把上层结点的结点号记到n中，把选中的相邻结点的结点号记到n1中。

在执行结点合并的过程中，总是把键值大的结点中的内容，合并到键值小的结点中去，以保证 1 号结点总是叶子首结点。为此，把要合并或均衡的两个结点的结点号记在变量 $n1$ 和 $n2$ 中，并保证 $n1 < n2$ 。在我们的程序中，同一层中的结点，编号小的结点里保存的键值也一定小，使结点合并更直观简便。

结点合并只出现在两个结点中的索引项数目之和小于 $ORDER$ 的情况下，即一个结点中的索引项数目为 $LIMITE$ ，另一个为 $LIMITE-1$ ，执行合并的几个语句很容易看懂，合并完成之后，要修改 $n1$ 结点中的索引项数目。

如执行的合并发生在叶子结点中，还必须修改结点间的指针链。如果删除的结点为叶子尾结点，还要修改叶子尾结点编号为 $n1$ 。

在本层结点中执行过结点合并之后，接着应在上层结点中执行新的删除操作，即把对



应于结点n2的索引项删除掉，并修改对应于结点n1的索引项的键值。接着检查剩在结点中的内容，以决定是否还要继续执行结点合并，或修改根结点编号。在我们的程序中，这些操作没有采用递归调用DELITE过程的方式来完成，而是在MERGE过程内完成，递归也是在MERGE过程之内进行的。如果改成在DELITE和MERGE之间彼此递归调用的方案来实现一次完整的删除操作，也是完全可行的。结束递归调用的条件，是在某一层结点中所剩的索引项数目不再小于LIMITE，或执行的删除已是在根结点中执行了。

当两个结点中的索引项数目之和大于或等于ORDER时，将执行结点内容的均衡操作。为了简便，我们采用的算法，是从选中的相邻结点中移一个索引项到刚执行了一次删除操作的结点中来，使该结点中的索引项数目等于LIMITE。正常的用法，多是真正在两个结点间执行均衡操作，也就是移动相邻结点中的若干个索引项（结点内实有索引项数目减掉LIMIT之后再除2得到的值），使两个结点中的索引项数目尽量接近。在执行这种结点均衡操作时，要按相邻结点是原结点的左邻还是右邻两种情况，进行不同的处理。为左邻时，要把它的关键值最大的索引项移到原结点的第一个索引项位置，原结点中原有的索引项都右移一个位置。为右邻时，要把它第一个索引项移到原结点原有索引项的右侧，并把该相邻结点中剩下的索引项都左移一个位置。当然，还必须修改两个结点中实有索引项的数目，并相应修改上层结点中有关索引项中的键值。

上页给出的即MERGE过程的程序流程框图。

程序中还有三个用于显示B⁺树的内容的过程TRACE，ACENDANT和DECENDANT。它们的功能和所用算法都比较简单，我们不再解释。

主程序部分被分为四个功能段，分别用于为变量设置初值、建立B⁺索引树、删除索引树中的某些索引项，然后把删除的索引项重新插入回去和显示全部叶子结点中的内容。在完成建树和某些删除操作之后，都用TRACE过程显示整棵树的内容。用户可以检查下面给出的结果，以加深对用B⁺实现索引表的原理和算法的认识。

我们还在几个关键过程中，用行注释的形式给出了几个调试语句。如果用户在阅读程序的过程中感到困难，可以把行注释符“!”去掉，重新编译该程序，这些调试语句将能显示程序运行过程中关键的中间结果，这对看懂程序中的有关算法，将提供很大的方便。

下面是这个程序的运行结果。

```

xxl= 50
blk= 12      3      16      11  32767
blk=  3      1      4       2      8      4      12      5      16
blk=  1      1      1       2      2      3      3      4      4
blk=  2      5      5       6      6      7      7      8      8
blk=  4      9      9     10     10     11     11     12     12
blk=  5     13     13     14     14     15     15     16     16
blk= 11      6     20      7     24      8     28      9     32
      10     36     13     40     14     44     15  32767
blk=  6     17     17     18     18     19     19     20     20
blk=  7     21     21     22     22     23     23     24     24
blk=  8     25     26     26     26     27     27     28     28
blk=  9     29     29     30     30     31     31     32     32

```

blk =	10	33	33	34	34	35	35	36	36
blk =	13	37	37	38	38	39	39	40	40
blk =	14	41	41	42	42	43	43	44	44
blk =	15	45	45	46	46	47	47	48	48
	49		49	50		50			

开始删除操作

blk =	3	1	7	2	15	6	23	7	31
	9	39	13	32767					
blk =	1	1	1	3	3	5	5	7	7
blk =	2	9	9	11	11	13	13	15	15
blk =	6	17	17	19	19	21	21	23	23
blk =	7	25	25	27	27	29	29	31	31
blk =	9	33	33	35	35	37	37	39	39
blk =	13	41	41	43	43	45	45	47	47
	49	49							

按升序显示全部叶子结点的内容

----acendant order output----

blk =	1	1	3	5	7	
blk =	2	9	11	13	15	
blk =	6	17	19	21	23	
blk =	7	25	27	29	31	
blk =	9	33	35	37	39	
blk =	13	41	43	45	47	49

按降序显示全部叶子结点的内容

----decendant order output----

blk =	13	49	47	45	43	41
blk =	9	39	37	35	33	
blk =	7	31	29	27	25	
blk =	6	23	21	19	17	
blk =	2	15	13	11	9	
blk =	1	7	5	3	1	

把 xx1 以内的正的偶数重新插入到树中

blk =	3	1	7	2	15	6	23	7	31
	9	39	13	43	16	32767			
blk =	1	1	1	51	2	3	3	52	4
	5	5	53	6	7	7			
blk =	2	54	8	9	9	55	10	11	11
	56	12	13	13	57	14	15	15	
blk =	6	58	16	17	17	59	18	19	19
	60	20	21	21	61	22	23	23	

```

blk=  7    62  21    25  25    63  26    27  27
      64  28    29  29    65  30    31  31
blk=  9    66  32    33  33    67  34    35  35
      68  36    37  37    69  38    39  39
blk= 13    70  40    41  41    71  42    43  43
blk= 16    72  44    45  45    73  46    47  47
      74  48    49  49    75  50
----acendant order output----
blk=  1    1    2    3    4    5    6    7
blk=  2    8    9   10   11   12   13   14   15
blk=  6   16   17   18   19   20   21   22   23
blk=  7   24   25   26   27   28   29   30   31
blk=  9   32   33   34   35   36   37   38   39
blk= 13   40   41   42   43
blk= 16   44   45   46   47   48   49   50

```

8.3 在用户的程序中使用索引文件

为了支持用户在自己的程序中能使用索引文件，就必须把在前一节给出的程序中实现的几个过程，以某种简单的方式提供给每个用户。比较理想的方案，是在PASCAL语言的一个UNIT中实现这些过程，并向用户提供调用的接口关系。而且，不应该苛求用户了解这些过程运行时所用的数据结构和实现原理，做到只按规定的接口关系就能完成这些调用。为此，应该把这些过程运行时内部使用的变量（与主程序无直接关系）全部放在UNIT的实现中，例如树的描述变量tree，树变量idx，索引路径变量path等等，主程序不必也不能直接使用它们。

采用在UNIT中实现这些过程，却要在另外的程序中调用它们的这种方案，与在前一节中给的程序例子是有某些区别的。主要表现在：

- 在UNIT实现中的某些变量，主程序是不能识别与访问的，例如tree.root，检索的路径变量path等。这就要求在变更前一节的程序为UNIT时，必须做某些必要的修改，例如：

原来的search过程用两个数值参数，一个是键值，一个是查找的索引块号。现在要变为一个数值参数，仍是键值，另一个为变量参数，是查找的结果，是search过程传给主程序的一个整型量。

因为search过程第二个参数用了变量参数，对inzert和delite过程中也是有影响的。原来开始的查找是通过直接给出的根结点号进行的，现在要用另外一个量，即UNIT的实现中给出的一个变量Q。

为了保证第一次查找一定从根结点开始，变原来的search过程为search1，在它的外层多用1个search过程。与此类似的，原来的trace过程变为trace1，在它的外层用上1个trace过程。

对树变量进行初始化的部分，变成一个init过程。

在完成插入的过程中，又多说明了一个变量X，用在该过程末尾部分。

• UNIT实现部分是用begin end.结束的，代替了原来的主程序部分。

为了精减文字，我们不在这里给出UNIT的实现部分的全部内容，只给出对上一节程序中变化了的有关部分，用……代表那些不加修改的部分。下面给出的是该UNIT的接口部分和实现部分的源清单。

```
INTERFACE,
unit btree( initial,search,inzert,delite,
             trace,acendant,decendant),

procedure initial;
procedure search(kx:integer; var no:integer);
    (* kx is key value,
       no is result searched *)
procedure inzert(kx:integer);
    (* kx is key value *)
procedure delite(kx:integer);
    (* kx is key value *)
procedure trace;
procedure acendant;
procedure decendant;

begin end;
    (* $include:'btree.inf' *)
implementation of btree;

const
    order= 8; limite= 4, tall= 5;
type
    item=record
        n, p: integer
    end;
    node=record
        ar:array [1..order] of item;
        link,rlink,layer,nb: integer
    end;
    treedsc=record
        root,tail,nextblk,nextdp: integer
    end;
    ar1=array [1..80]of node;
    ar2=array [0..tall] of record
        blk,loc,dp: integer
    end;
```

```

var
  tree:treedsc;
  lay,blk,q: integer;
  idx: ar1;
  path: ar2;
  splitdo:boolean;

procedure initial;
begin
  with tree do [root:=1;      nextblk = 2;
                tail:=1;      nextdp:=1];
  splitdo:=false;
  with idx[1]do
    [nb:=0; blk:=0; rln:=0; layer:=0]
end;

procedure search;
  procedure search1(kx, blk:integer);
    var i:integer;
  begin
    with idx[blk]do
      [ i:=1;
        while (kx>ar[i].n) and (i<=nb) do i:=i+1;
        path[layer].blk:=blk;
        path[layer].loc:=i;
        if(i<=nb) and (kx=ar[i].n)
          then path[layer].dp:=ar[i].p
          else path[layer].dp:=0;
        if layer < 0
          then search1(kx,ar[i].p)
          else no:=path[layer].dp
        ]
      end;
  end;

begin (* search body *)
  search1(kx,tree.root);
end;

procedure insert;
  var i, k, oldblk, newblk, j: integer;
  genroot: boolean;
  x: integer;

```

```

      . . . . .
begin      (*inzert procedure body*)
  genroot := false;
  if not splitdo
    then[search(kx, q), lay:= 0],
  if not splitdo and(path[0].dp(>0))
    then[writeln('? This number exists, no inserted'),
          writeln(path[0].dp)]
  else
    [oldblk := path[lay].blk;
     i := path[lay].loc;
     . . . . .
     if splitdo then
       [ lay := lay+1;
         j := path[lay].blk ; i := path[lay].loc;
         if newblk < > tree.tail
           then [idx[j].ar[i].n
                  := idx[oldblk].ar[limite].n,
                  path [lay].loc:=path[lay].loc+1,
                  bk := newblk;
                  x:=idx[bk].ar[idx[bk].nb].n ]
           else[idx[j].ar[i].p:=newblk;
                bk := oldblk;
                x := idx[bk].ar[idx[bk].nb].n ],
              insert(x)
            ]
        ]
  end,
  . . . . .
begin      (*delite procedure body*)
  tailnode:=false ;
  search(kx, q), lay:= 0;
  if path[lay].dp=0
    then
      writeln('? This number no exists, no deleted')
      . . . . .
procedure trace;
procedure trace1(blk:integer);
  va: is integer;

```

```

begin
  with idx[blk] do
    [writeln('blk=', blk: 3);
     for i:=1 to nb do
       [write(ar[i].p: 8, ar[i].n: 6) ;
        if(nb> 4) and (i= 4) then writeln] ;
     writeln;
     for i:=1 to nb do
       if layer( )> 0 then trace1(ar[i].p)
     ]
  end;

begin          (*trace body*)
  trace1(tree.root)
end;

. . . . .

begin
end.

```

有了上述的UNIT，我们就可以在自己的程序中使用它。作为应用UNIT的通用办法，在程序最前端要用注释形式给出\$include编译命令，把UNIT的接口内容引入到自己程序中。在程序标题之后，用USES子语句给出使用UNIT各过程的过程名。在我们的程序中没有用到trace过程。应该说，该UNIT的最后三个过程用于程序调试目的，并无实用意义。

该程序的功能，是建立六名人员简单情况（职工编号，姓名和年龄三项内容）的一个数据文件，并用索引功能管理职工编号，执行了插入、检索、删除功能，最后用升序和降序关系显示了索引树中的内容。

这个例子是原理性的，表明索引文件的一般用法。要达到实用的水平，还必须解决另外许多问题。有些问题，要在上述的UNIT中解决，另一些问题，涉及到程序和UNIT的接口关系，我们将在下一节对这些问题进行某些必要的说明。

下面是该程序的源程序清单和程序运行时屏幕上显示的入/出操作的内容。

```

(* $include: 'btree.inf' *)
program btr(input, output);

  uses btree(initial, search, insert, delete,
             trace, acendant, decendant);

type
  str=string(10);
  rec=record
    key: integer;
    name: str;
    age: integer

```



```

        end;

var
    k, i, l, xx, xxi: integer;
    nam: str;
    ag, ke: integer;
    f: file of rec;

procedure position y, x: integer;
begin
    write(chr(27), '[', y div 10:1, y mod 10:1,
           ', ', x div 10:1, x mod 10:1, 'H')
end;

begin
    (* main program *)
    assign(f, 'a.dat');
    f.trap:=true;
    f.mode:=direct;

    (* part 1: initialize the idx-tree *)
    initial;

    (* part 2: create the index-tree *)
    (* and write data into file f *)
    rewrite(f);
    write(chr(27), '[J');
    position(8,10); write('key    name    age');
    for i:=1 to 6 do
        [position(9+i, 6); write(i:3, chr(27), '[K' ;
        position(9+i, 12); readln(ke);
        position(9+i, 18); readln(nam);
        position(9+i, 32); readln(ag);

        f^.key:=ke;    f^.name:=nam;    f^.age:=ag;
        insert(ke);    put(f)
        ],
    writeln;

    (* part 3: search a record by using key *)
    write('which key value for searching ? ');
    readln(ke); search(ke, k);
    if k < > 0 then
        [ write('searched record content : ');

```

```

        seek(f, k); get(f);
        writeln(f^.key: 4, f^.name: 14, f^.age: 4)];
    writeln;

    (* part 4: delete an item in the index-tree *)
    write('delete record key= ');
    readln(k);
    delite: k);
    writeln;

    (* part 5: display the contents of all leaf nodes *)
    acendant;
    decendant
end.

```

程序运行的结果:

	key	name	age
1	12	aaaaaa	20
2	5	bbbbbb	12
3	40	ccccccc	35
4	7	ddd	21
5	100	eeeee	60
6	30	f	36

which key value for searching ? 40
 searched record content: 40 cccccccc 35

delite record key= 5

---acendant order output---			
key	7	dp	4
key	12	dp	1
key	30	dp	6
key	40	dp	3
key	100	dp	5
----decendant order output----			
key	100	dp	5
key	40	dp	3
key	30	dp	6
key	12	dp	1
key	7	dp	4

8.4 实现与使用索引文件中的实际问题

前两节的内容是原理性的，在使用索引文件解决实际应用问题时可不这么简单，还有许多实际问题必须加以处理，其中主要包括：

(1) 索引文件中的键值域可以取不同的类型，如字符串类型，整数类型、字类型等等，还可能由几个域（相同或不同的类型）组合成一个键值，键值域的长度也是可变的（如不同长度的字符串，整数或长整数等等），管理索引结构的程序（上一节UNIT中实现的过程等）要能统一正确地协同处理，而我们上节例子中只处理了长度为2的整数一种类型。

(2) 对同一个数据文件，可能用到几个不同的索引键。通常情况下，一定有一个能唯一地标识一个记录的主键（键值不允许有重值，即任何两个记录之间都不能有相同的主键值），还可能有零到多个副键，副键既允许有重值出现，也可以规定不允许重值，由用户决定。为此，必须有办法描述并记录每个用到索引结构的数据文件的记录组成情况和索引键组成情况，还必须处理好多个键之间的联系。通常情况下，每个键都要有单独的一棵索引树。

(3) 从索引文件的读写效率考虑，最好给出适当的内存缓冲区，存放正在使用的索引文件当中的某些内容，而不是每读写一个数据记录都要多次读写磁盘。

还有其它一些实际问题，这里不再说明。

第九章 和汇编语言程序的接口关系

9.1 问题的提出

PASCAL是高级程序设计语言，它缺乏汇编语言所能提供的某些功能，特别是它不能直接提供使用汇编语言中中断调用的手段，这就使得PASCAL语言的用户无法在自己的程序中使用操作系统提供的功能调用，也就不能了解诸如所用微机的操作系统的名称和版本号、系统中配备的设备和相应时刻设备的使用情况和状态等与系统有关的信息，也不能直接了解内存的使用情况和按用户的意图分配内存，还不能在自己的程序中完成诸如分析磁盘上文件目录的内容，改名、复制和删除文件等操作。初学PASCAL语言的读者会觉得提出这些问题是过分要求，而实际上，在许多大型实用程序中，例如在数据库管理系统中，是一定要能在程序中直接了解上述信息和进行上述各有关操作的。解决的办法是，在PASCAL的程序中调用用汇编语言设计好的子程序。这些子程序可以做成PASCAL程序中的外部过程或外部函数的形式，可以带形式参数，也可以不带形式参数。换言之，对PASCAL编译程序未能提供的、而实际应用中又必然用到的系统功能调用，PASCAL语言的用户可以通过少量的汇编子程序实现出来，然后在自己的PASCAL程序中就能使用它们。在本章中，我们将提供实现汇编语言子程序的有关基础知识，以及处理汇编语言与PASCAL语言程序之间的接口等有关内容。

9.2 汇编语言程序设计的简明知识

我们无意在本节对微机上的汇编语言程序设计做过多的介绍，这些知识在专门讲解汇编语言的书籍与资料中有详细说明，我们只想在这里给出其中对我们最有用的某些简明内容。对有些内容将简单说明几句，对另一些内容只给出范例。读者务请注意，这些内容既不系统更不完整，只供读懂本章中有关汇编语言的程序或子程序作参考用。

IBM-PC/XT和WANG-IPC机的CPU都是用INTEL8086实现的。

一、8086和8088芯片中的寄存器

芯片中有13个用户能访问的寄存器，它们是：

IP 指令计数器

AX, BX, CX, DX 四个通用寄存器

其中A, B, C, D后面可以跟X, H, L, 分别代表完整的寄存器，
寄存器的高位字节和寄存器的低位字节

SP 栈指针寄存器

BP 基地址指针寄存器

SI 源索引寄存器

DI	目的索引寄存器
CS	代码段基地址寄存器
DS	数据段基地址寄存器
SS	栈段基地址寄存器
ES	外部数据段基地址寄存器

二、五种寻址方式

(1) Direct offset Addressing 直接偏移寻址

```
MOV     AX, word 1
MOV     AX, BX
```

(2) Indirect Addressing 间接寻址

```
MOV     AX, [BX]
MOV     [BX], AX
```

(3) Register Offset Addressing 寄存器偏移寻址

```
MOV AX, [BP + 2]
```

(4) Addressing through Base and Index Register

```
                [BX][SI]
MOV AX, [BX][DI]    通过基地址和变址
                [BP][SI]    寄存器实现的寻址
                [BP][DI]
```

(5) Addressing through Base and Index Register

Plus an offset

```
                [BX][SI]
MOV AX, VAR1    [BX][DI] 通过基地址和变址寄存
MOV AX, [BX]    [SI + 10] 器加偏移实现的寻址
MOV AX, VAR1    [BP][SI]
                [BP][DI]
```

三、状态标志位

CF	进位
AF	附加进位
OF	溢出位
ZF	结果为零位
SF	符号为负位
PF	奇偶校验

控制标志位

DF	处理字符串时
----	--------

0 按从低向高字符进行
 1 按从高向低字符进行
 IF
 0
 1 允许中断
 TF
 0
 1 Trap (陷阱) 进入单步方式

四. 源程序的构成

合法字符 A--Z a--z 0--9 ? . @--\$
 ' ' " " ; , [] () ,

名字以字母开头, 最多31个有效字符

程序 { Name PROG2
 :
 START: MOV AX,ABC
 :
 END START

段 { SA segment [align] [combine] ['class']
 :
 : PARA PUBLIC
 : BYTE COMMON
 : WORD AT expression
 : SA ENDS PAGE STACK
 :
 : MEMORY

或空白, 即无组合特性, 是自用段

Assume语句把段的基地址赋给一个段寄存器名字。如果没有使用Assume语句, 则在每次使用一段之前, 就必须在段内的符号之前补上段名前缀。

五. 语句行的格式

伪指令 名字 动作 表达式; 注释
 指令 标号 活动 表达式; 注释

命令语句的成分:

LABEL, MNEMONIC argument, ..., argument, comment

命令行是由多种成分组成的: 变量、标号、常量, 数字, 注释和表达式。表达式又是由操作数, 操作符、串和各种符号组成的。

(1) 变量 是用一个符号名代表的一个数据元素。它有三个属性: 类型, 所在段和

段内偏移。

变量类型		变量属性
DB	字节	所在段
DW	字	段内偏移
DD	双字	
DC	八字节	
DT	十字节	
Struct	结构	
RECORD	记录	

(2) 标号同样有三个属性，所在段、偏移和类型。段属性是指定义该标号的段的开始的节数 (paragraphe number)。标号总是被指定给 CS 寄存器。类型是被分为 NEAR 或 FAR 两种。属于 NEAR 类型的标号只能在相同段或半中被访问，而属于 FAR 类型的标号还可以被其它段访问。PASCAL 程序要引用的汇编子程序就以被定义为 FAR 类型的 PROC。

(3) 数和它们的记法

0101011B	二进制数	2.21E-7	浮点实数
77.0或735Q	八进制数	8217R	实数
9.51或951D	十进制数		
0FFH	十六进制数		

(以字母开头的数应以一个 0 字符开头才不会误会)

(4) 串——用单引号引起来的一些字符。

'a string'

'goodbye'

(5) 常量——有确定的值，分为整型常量和串常量。

(6) 注释——用于给出说明性的文字

, ... COMMENT' ...

, ... 或'

,'

(7) 伪指令——用于向汇编程序表明：

输入/输出

格式控制

内存组织

条件汇编

汇编清单和交叉访问控制

各种定义

(8) 操作数——右为源、左为目的的操作数

立即数	MOV	AX	9	
寄存器	MOV	AX	BX	
内存数	MOV	AX	VAR1	直接
	MOV	BX	[VAR1]	间址

MOV BX [VAR1+5] 变址

(9) 属性操作符

SEG	得到标号或变量的段的基地址
OFFSET	得到标号或变量的偏移地址
TYPE	得到变量的字节数
LENGTH	得到DUP项操作对变量赋初值的单元数，对其它则回送1。
SIZE	给出 LENGTH和TYPE的乘积。
• TYPE	按照变量的定义返回字节数。
SHORT	如果JMP(转移指令)的转移距离在-128到127之间，用它修改JMP的NEAR属性，可使目标机器指令缩短一个字节。
PTR	用BYTE,WORD或DWORD废除变量定义的类型(DB,DW或DD)，用NEAR或FAR废除标号的距离属性。
WIDTH	返回一个记录或一个记录域的宽度值。

(10) 数据伪指令

数据伪指令用于定义数据类型和给出数据的初值。可以用它：分配数据空间；为数据赋值；设置或重新定义常量或标号；建立数据结构；定义一个名字的属性；变换默认的数字基值；变化地址分配的位置计数。

```

Assume
DB
DD
DQ
DT
DW
EQE          • RADIX
-
EVEN          RECORD
INCLUDE      STRUC...ENDS
LABEL
NAME
ORG
PROC...ENDP
DB 2         一个字节，初值为2
DT ?         10个字节，不赋初值
DB PTR DD LABEL
              保留地址
MULT-CHAR DB'MARC MILE PAUL'
DW 100 DUP (0, 1)
              200个字，其初值交替为0为1
DB 1,2,3     三个字节，初值分别为1,2和3
DB 2,4,7 , 02FH

```


INCLUDE RECORD.TST 包含源程序文件 RECORD.TST (在默认盘上)

```
VAR1 LABEL NEAR  
VAR2 LABEL FAR
```

} 标号伪指令

PROCEDURE 是一段指令序列, 用于把一个程序分成若干模块, 各自完成一个特定的功能。CALL语言可以调用一个PROCEDURE, PROC内的RET实现返回。PROC...ENDP

例

```
procedure name PROC [FAR/NEAR]  
:  
:  
RET  
:  
procedure name ENDP
```

记录伪指令 一个记录由16个二进制位组成

rec1 RECORD HIGH:4=1, MID:3=2, LOW:3=5

结构伪指令 一组字节, 字等组成的数据

```
cube STRUC  
L DW ?  
W DW ?  
H DW ?  
cube ENDS  
SQUARE cube <1,2,3>  
S STRUC  
FIELD1 DB 1,2  
FIELD2 DB 10 DUP (?) } 不能重新赋值的域  
FIELD3 DB 5  
FIELD4 DB 'JOHN DOE' } 可以重新赋值的域  
S ENDS  
NEW S<,,7, 'BOB HALLER'>
```

多出字符会被切掉

访问时

```
S.FIELD1  
S.FIELD4 等等  
LONG-NECK Z00<16,5,7>  
MOV AL, LONG-NECK.Giraffe  
Z00 STRUC  
Giraffe DB?  
Zebra DB?  
Gazelle DB?  
Z00 ENDS
```

六、DOS的中断和功能调用

DOS保留中断20H~3FH内部使用。内存的绝对地址80h~fffh用于这些中断，其功能分配如下：

- 25 绝对盘地址读
- 26 绝对盘地址写
- 24 致命错误夭折
- 21 功能调用
- 20 程序结束
- 23 SHIFT+CANCEL退出地址
- 27 程序结束，但代码留驻内存。

以下仔细阐述：

- 20 程序结束 文件缓冲区内容→disk，不关闭文件时，文件长度如果有变化，目录中不能反映这个变化情况，即文件长度保持不变。执行本中断，CS必须指向100H的参数域
- 21 操作系统功能调用 通过该中断可直接调用操作系统的许多功能模块，如文件管理、输入输出等。
- 22 程序结束地址（存放在单元0:88H~8AH） 当一个程序结束时，控制转向该结束地址。在程序段建立时，该结束地址被复制到程序段前缀PSP中。当一个程序要运行一个子程序而且希望子程序（不是子过程）结束后控制返回到父程序中，则先用功能调用X'25'来设置返回地址，否则，子程序结束时也终止了父程序。
- 23 CTRL-Break出口地址 当一个程序运行过程中检查到中止键符CTRL-Break时，则执行该号中断，使程序控制转向该出口地址。这个地址在程序段建立时被复制到PSP中，一个程序退出时，会恢复该地址值。若希望一个程序接收到CTRL-Break中止键，程序转向某个特定的处理程序，可用系统调用X'25'将该处理程序的入口地址填入该向量单元中。
- 24 如果DOS产生一个致命错误，将执行该中断。这个地址在程序段建立时被复制到PSP中。该中断结束时DI低字节返回错误代码。
- 25 绝对磁盘读。入口参数为：
AL=驱动器号（0=A；1=B，等）；
(CX)=要读的扇区数，(DX)=起始逻辑扇区号，
DS:BX=磁盘传送地址区。
中断执行的结果将磁盘上的扇区内容读入内存。传输成功，清标志位CF，否则置CF=1，同时AL中返回错误代码。
- 26 绝对磁盘写，入口参数设置同上，结果返回值也同上，只是为写磁盘。
- 27 程序结束，但驻留内存。该中断执行后程序内容不会被其它程序调入时覆盖。用该中断，程序驻留长度不能超过64K，输入参数为DX=程序最末地址+1。

下面是DOS提供给用户选择使用的子程序表。

用户调用子程序的方法是：把该子程序的编号放入寄存器AH中，按表中要求设置入

口参数，然后执行软件中断21H。之后可根据返回的结果进行分析。

对DOS 2.0版，AX寄存器中返回错误码或返回值（如果CF=0，那么AX中返回为错误码）。各错误码对应含义如下：

0 = 正常返回，无错误发生。

1 = 非法的功能调用（如入口参数设置不合法等）；

2 = 文件没找到。

3 = 路径没找到。

4 = 打开文件太多。

5 = 不允许访问。

6 = 文件号非法。

7 = 内存块有错。

8 = 没有足够的内存区。

9 = 非法块。

A = 环置设置出错（对4BH号而言）。

B = 格式有错。

C = 访问不合法。

D = 数据不合法。

F = 驱动器号不合法。

11 = 没有该种设备。

BIOS 功能调用分配表

编 号	功 能	入 口 参 数	出 口 参 数
01H	键盘输入字符		AL = 输入字符
02H	显示器输出字符	DL = 输出字符	
03H	串行设备输入字符		AL = 输入字符
04H	串行设备输出字符	DL = 输出字符	
05H	打印机输出字符	DL = 输出字符	
06H	直接控制台 I/O	DL = FF (输入) DL = 字符 (输出)	AL = 输入字符
07H	直接控制台输入 (无回显)		AL = 输入字符
08H	键盘输入字符(无回显)		AL = 输入字符
09H	显示字符串(以 '\$' 结 尾)	DS:DX = 缓冲区首址	
0AH	输入字符串	DS:DX = 缓冲区首址	
0BH	检查标准输入状态		AL = 00 无键入 AL = FF 有键入
0CH	清输入缓冲区并执行 指定的标准输入功能	AL = 功能号 (01, 06, 07, 08 或 0a)	

(续表)

编 号	功 能	入 口 参 数	出 口 参 数
0DH	初始化盘状态		
0EH	选择当前盘	DL = 盘号	AL = 系统中盘的数目
0FH	打开文件	DS:DX = FCB 首址	AL = 00 成功 AL = FF 未找到该文件
10H	关闭文件	DS:DX = FCB 首址	AL = 00 成功 AL = FF 已换盘
11H	查找第一个目录项	DS:DX = FCB 首址	AL = 00 成功 AL = FF 未找到
12H	查找下一个目录项	DS:DX = FCB 首址	AL = 00 成功 AL = FF 未找到
13H	删除文件	DS:DX = FCB 首址	AL = 00 成功 AL = FF 未找到
14H	顺序读一个记录	DS:DX = FCB 首址	AL = 00 成功 AL = 01 文件结束 AL = 03 缓冲不满
15H	顺序写一个记录	DS:DX = FCB 首址	AL = 00 成功 AL = FF 盘满
16H	建立文件	DS:DX = FCB 首址	AL = 00 成功 AL = FF 目录区满
17H	文件更名	DS:DX = FCB 首址 (DS:DX + 17) = 新名	
18H	无		
19H	取当前盘盘号		AL = 盘号
1AH	置磁盘缓冲区	DS:DX = 缓冲区首址	
1BH	取当前盘的文件定位表 (FAT) (3.0版以上无)		DS:BX = 盘类型字节地址 DX = FAT 表项数 AL = 每簇扇区数 CX = 每扇区字节数
1CH	取指定盘的文件定位表 (3.0版以上无)	DL = 盘号	同1BH
21H	随机读一个记录	DS:DX = FCB 首址	AL = 00 成功 AL = 01 文件结束 AL = 03 缓冲不满
22H	随机写一个记录	DS:DX = FCB 首址	AL = 00 成功 AL = FF 盘满
23H	取文件长度	DS:DX = FCB 首址	AL = 00 成功 AL = FF 未找到
24H	置随机记录号	DS:DX = FCB 首址	
25H	置中断向量	AL = 中断类型号 DS:DX = 入口地址	
26H	建立一个新的程序段	DX = 段号	
27H	随机读若干记录	DS:DX = FCB 首址 CX = 记录数	AL = 00 成功 AL = 01 文件结束 AL = 03 缓冲不满

(续表)

编 号	功 能	入 口 参 数	出 口 参 数
28H	随机写若干记录	DS:DX=FCB首址 CX=记录数	AL=00成功 AL=FF盘满
2AH	取日期		CX:DX=日期
2BH	置日期	CX:DX=日期	
2CH	取时间		CX:DX=时间
2DH	置时间	CX:DX=时间	
2FH	取磁盘缓冲区首址		ES:BX=缓冲区首址
30H	取DOS版本号		AL=版本号; AH=发行号
31H	终止用户程序并驻留内存	AL=退出码 DX=程序长度	
33H	置/取CTRL-BREAK 检查状态	AL=00取状态 AL=01置状态DL	DL=状态
35H	取中断向量	AL=中断类型	ES:BX=入口地址
36H	取磁盘空闲空间大小	DL=盘号	BX=未分配单元数目; DX=已分配的单元数目; CX=每簇中的字节数; AX=每一分配单元的簇数;
39H	建立了目录	DS:DX字符串首址	
3AH	删除子目录	DS:DX字符串首址	
3BH	改变当前目录	DS:DX=字符串首址	
3CH	建立文件	DS:DX=字符串首址 CX=文件属性字	AX=文件号
3DH	打开文件	DS:DX=字符串首址 AL=0读 AL=1写 AL=2读/写	AX=文件号
3EH	关闭文件	BX=文件号	
3FH	读文件	BX=文件号 CX=读入字节数 DS:DX=缓冲区首址	AX=实际读入的字节数
40H	写文件	BX=文件号 CX=写盘字节数 DS:DX=缓冲区首址	AX=实际写入的字节数
41H	从指定目录中删除一个文件	DS:DX=包含驱动器名、路径名和文件名的ASCII字符串地址	

(续表)

编 号	功 能	入 口 参 数	出 口 参 数
42H	改变文件读写指针	BX=文件号 CX:DX=位移量 AL=0绝对移动 AL=1相对移动 AL=2绝对倒移	DX:AX=新的指针位置
43H	置/取文件属性	DS:DX=字符串地址 AL=0取文件属性 AL=1置CX=属性	CX=文件属性
44H	设备文件I/O控制	BX=文件号 AL=0取状态 AL=1置状态DX AL=2读数据 AL=3写数据 AL=6取输入状态 AL=7取输出状态	DX=状态
45H	复制文件号	BX=文件号1	AX=文件号2
46H	强制复制文件号	BX=文件号1 CX=文件号2	CX=文件号1
47H	取当前目录路径名	DL=盘号 DS:SI=字符串地址	DS:SI=字符串地址
48H	分配内存空间	BX=申请内存数量	AX:0=分配内存首址 BX=最大可用内存空间 (失败时)
49H	释放内存空间	ES=内存空闲块始址	
4AH	修改已分配的内存空间	ES=原内存始址 BX=再申请的数量	BX=最大可用空间 (失败时)
4BH	装入一个程序	DS:DX=字符串地址 ES:BX=参数区首址 AL=0装入执行 AL=3装入但不执行	
4CH	终止当前程序并返回到调用程序	AL=退出码	
4DH	取当前程序的退出码		AX=退出码

9.3 在 PASCAL 程序中调用汇编语言的子程序

在PASCAL程序中调用汇编语言的子程序，是一项很实用的技术，是扩展PASCAL语言功能的重要手段，常常被用来实现那些在汇编语言中能完成、而PASCAL语言却不能支持的程序功能，特别是微机系统中的中断调用功能。我们将在本节详细讲解实现这种调用的有关概念和有关技术。

为了实现上述调用，必须从PASCAL程序（主调用者）和汇编语言子程序（被调用者）两个方面处理好它们之间的关系，主要是过程名（或函数名）、过程或函数的参数识

别及访问、以及函数结果的返回三个问题。在详细讲解这些问题之前，先看一个实际例子，是 PASCAL 程序调用用汇编语言实现的一个过程 CURSORPOS，引用一个函数 KEYIN 的具体应用。

PASCAL 程序（文件名为 P.PAS）的主要功能是：

- 把光标定位到终端屏幕的第10行第20列的位置，然后调用CURSORPOS过程取得光标当前位置的行、列值（正确结果当然应为10和20），并把该值显示在屏幕上，执行回车、换行后，再测光标位置并再次显示。这一段用于验证测定光标位置的过程CURSORPOS运行的正确性。

- 用户按终端键盘上任意的一个键，引用函数KEYIN读来该键的扫描码和ASCII码的值，并把这两个值显示到终端屏幕上。这一段给出了识别打入的每一个键的具体方法，也用于验证取得所击键的编码值的函数KEYIN运行的正确性。

- 整个程序是用REPEAT语句控制的，仅在用户输入问号字符时才结束。这样做的目的是允许用户用这个程序了解键盘上每个键的扫描码的值和ASCII码的值。所以这个程序在作为 PASCAL 程序调用汇编语言子程序实例的同时，也为读者提供了一项很实用的程序功能。程序中的I用来对REPEAT语句执行的次数进行计数。

下面是该程序的源程序清单。

```
program cursor (output) ; (* P.PAS *)
var row, column: integer;
    i: integer; code: word;
    asc, scan: byte;
function keyin: word; extern;
procedure cursorpos (var y, x: integer) ;
    extern;
begin
    i := 0;
    repeat i := i + 1;
        write (chr (27) , '[10; 20H') ;
        cursorpos (row, column) ;
        writeln ('Cursor position:', row:4, column:4, i) ;
        cursorpos (row, column) ;
        writeln ('Cursor position:', row:4, column:4) ;

        write ('Enter any key! ') ;
        code := keyin;
        scan := hi byte (code) ;
        asc := lo byte (code) ;
        writeln (' The key scan code and ascii code',
            scan:4, asc:4) ;
    until asc = wrd ('? ')
end.
```

PASCAL 程序中说明和使用的外部过程CURSORPOS和外部函数KEYIN，是在文

件名为 A.ASM 的汇编程序中实现的，用汇编语言实现它们的主要意图是完成系统提供的中断调用。PASCAL 语言本身尚不支持这种调用，只能用汇编语言来完成。每个汇编语句的功能都以注释形式跟在语句后边，整个过程和函数的详细说明将在稍后部分讲解。

下面就该汇编程序的源程序清单。

```

; A.ASM
; keyborad input subroutine FUNCTION
assume             cs:cseg
cseg              segment 'CODE'
public    keyin
keyin proc        far                ; 定义函数
    push         bp
    mov          bp,        sp
    mov          ah,        0        ; 执行16h的 0 功能码
    int          16h                ; 的中断调用
    pop          bp
    ret
keyin endp
;
public cursorpos
cursorpos proc    far                ; 定义过程
    push         bp                ; 原BP寄存器的值进堆栈保留
    mov          bp,        sp      ; 栈指针的值写入BP
    mov          bh,        0
    mov          ah,        3        ; 执行10h的3功能码
    int          10h                ; 的中断调用

    xor          ax,        ax      ; 把ax寄存器清零
    mov          al,        dh      ; dh:row number
    inc          ax                  ; 把光标的行位置的
    mov          si,        [bp+8]  ; 值写入堆栈相应单元
    mov          [si],      ax
    xor          ax,        ax
    mov          al,        dl      ; dl:col number
    inc          ax                  ; 把光标的列位置的
    mov          si,        [bp+6]  ; 值写入堆栈相应单元
    mov          [si],      ax
    pop          bp
    ret          4
cursorpos endp
cseg ends

```


end

为了得到由上述两部分内容形成的可执行的目标程序，具体操作过程如下：

PAS1 P, 对P·PAS文件进行编译

PAS2

ASM A, 对A·ASM文件进行汇编

LINK P+A, 对两个OBJ文件进行连接

P 执行得到的P·EXE目标程序

下面我们将结合上述实例，详细讲解PASCAL程序调用汇编语言子程序必须解决的三个问题。

(1) 过程（或函数）名处理

在PASCAL程序中，必须把用汇编语言实现的过程或函数说明为自己的外部过程或外部函数，也就是在过程、函数的首部的末尾加上EXTERN命令，这是保证正常编译和后来的连接操作所必须的。在汇编语言的程序中，必须把所实现的过程、函数名说明为PUBLIC属性。这里说的过程或函数名，实际上是汇编子程序入口的那条汇编指令的标号，是实现过程调用的程序段的入口地址，上述例子中的KEYIN和CURSORPOS就属于这种情况。

(2) 过程、函数的参数处理

在PASCAL程序中，说明与使用用汇编语言实现的过程或函数的参数，和说明与使用PASCAL语言本身的过程或函数的参数的办法完全相同，问题仅体现在怎么在汇编语言的子程序中访问这些参数。要透彻地讲明这种联系，需要了解PASCAL编译程序对过程、函数的编译办法，以及处理过程调用语句、引用函数等的具体技术，这些都超出了本书的讨论范围。我们只准备就与具体程序设计有关的几个问题，给出某些扼要说明。

在PASCAL程序中，说明与调用过程都由PASCAL编译程序处理好了，没给我们留下任何要另行处理的问题。在第五章讲解过程与函数时，已说到程序要调用一个过程，首先要为它在堆栈中分配一些单元，用来保留返回的断点，给出被调用过程的入口的地址，保留被调用过程的局部变量和它的参数（如果存在的话）等。被调用过程运行结束后，要收回它刚才占用过的堆栈区域。作为结论，我们可以这样说：

- 如果过程或函数带有参数，在它们投入运行之前，调用者已为它们的参数分配好堆栈区，并把返回地址也放入堆栈。在汇编语言子程序中（过程或函数），只能通过这些参数在堆栈中的位置来找到和使用这些参数。堆栈是PASCAL程序和它的汇编语言子程序进行数据通讯（以参数形式）的公用数据区。

- 过程、函数的变量参数，直接给出的是该参数的地址，而值参数直接给出的是参数的值。这两种参数的初值，是由主调用者在调用时给在堆栈相应单元中，被调用者（过程或函数）可以访问这些单元。在汇编语言的子程序中，必须以间接寻址方式访问变量参数，例如上例中的MOV si, [bp+6]和MOV [si], ax两条指令所实现的，就属于这种情况。如果在PASCAL程序中把CURSORPOS过程的两个参数说明为值参数，在汇编子程序中就可以用MOV ax, [bp+6]的指令把该参数的值取到ax寄存器中。

- 参数的个数和每个参数占用的存储单元数、参数在内存中的存放方法，设计汇编语言子程序的人员必须十分了解，并能正确地操作它们。如果出现使用不当的情况，就不能得到正确的程序结果。在我们上述的例子中，用的都是一个字的整数或字类型的数，正好

能放在一个内存字或一个通用寄存器中，属于最简单的情况。

• 在汇编语言子程序结束运行，返回主调用者的断点时，要在子程序中修改堆栈指针，使其指向第一个参数下面的位置。这是通过在ret指令中给出一个字节数的办法完成的。该字节数的计算值，是通过计算参数实际占用的字节数得出的。例如，上例中函数不带参数，ret不带任何值（相当于零），而过程带两个字类型的参数，占用4个字节，返回指令要写成ret 4的形式。当这个字节值给错时，堆栈使用将出现错误，程序不能正确运行。

（3）函数值的返回处理

PASCAL程序中，过程的说明和函数的说明是不同的。在引用函数时，主引用者将把函数名作为指定类型的一个变量分配在堆栈中，并在函数运行结束时从那个（些）单元取得函数的返回值，与调用过程的情况不完全一样。

有了上述说明，我们就可以着手介绍实现汇编语言子程序的具体技术。

用汇编语言实现PASCAL程序的外部过程或函数，主要涉及到，

怎样定义这些过程或函数；

怎样解决过程名、函数名的连接问题；

怎样正确找到与使用堆栈中的参数；

怎样解决过程、函数的返回问题。

这些问题都是针对PASCAL程序和它的汇编语言子程序的衔接关系提出来的，而不是讲有关纯汇编语言程序设计的知识。

在汇编语言中，定义其它程序模块中（包括汇编语言的程序）要使用的过程或函数，都用PROC FAR指令实现，这里的PROC代替过程或函数，FAR表示在其它程序模块中可引用该过程或函数。通常过程名或函数名要用作该指令的标号，并给它以PUBLIC（公用）属性，以保证执行连接时，连接程序能找到这个名字。

汇编语言子程序是用RET指令结束运行并实现返回的，RET后可能跟随的一个字节数n，用于修改堆栈指针，即实现sp的值增量n，前边已讲到它的作用。

找到与使用堆栈中的参数的关键，是合理使用BP寄存器（基地址指针）。为了在相应子程序中使用BP，必须把原来BP的值记忆下来，用PUSH BP指令实现。在子程序结束时，再恢复原BP值。这就保证了主调用者程序的BP值不被破坏。在子程序中使用参数是通过进入该子程序时的堆栈指针SP加一个偏移量进行的，而SP是会变化的，为此，把原SP的值送入BP，用BP加相应偏移量的办法访问堆栈中参数。

下页图是刚进入子程序时和执行PUSH BP指令之后的堆栈中的情况（虚线所示部分）。执行过PUSH BP之后，栈顶中是BP的原来值，MOV bp, sp指令使BP也指向栈顶。栈顶之下是用两个字表示的返回地址（段号和偏移值各占一个字）。再下面是按从后向前的次序保存的过程参数。因此，过程的最后一个参数保存在BP+6（参数用的是一个字长时）单元，依据每个参数实际占用的单元数，就可以求得每个参数在堆栈中的实际位置。汇编语言子程序就是通过这个位置来找到与使用堆栈中的每一个参数的。我们再次提请读者注意，要用间址寻址方式使用变量参数。对值参数，相应位置中保存的就是该参数的值，可以是主调用者提供的（复制来的实参值），也可以是在子程序内运算得到的，子程序内修改了的值参数，只保存在堆栈相应单元中，对主调用者所用的值型实参不会产生影响。而变量型参数的读写都是通过间址操作在实际参数上进行的。用VAR说明的变量参数总是占用堆栈中的一个字，存放的是变量地址，与参数本身的类型无关。

上述汇编语言子程序中用到两断中断调用。
 16h的（这里的h代表16进制）子功能码为0的中断，完成取得在键盘上所按的一个键的扫描码和ASCII码，其结果放在dx寄存器中。高位字节放的是扫描码，低位字节放的是ASCII码。除个别键（如CTRL，SHIFT，ALT等）外，每个键都有自己的扫描码，它的值取决于键在键盘上的位置。而全部可打印字符（包括空格）、CTRL键与英文字母键的复合、ESC键等还有自己的ASCII码编码值，而全部的功能键，编辑专用键则只有自己的扫描码，而无相应的ASCII编码值，此时在DX寄存器的低位字节上给出的是零值，我们正是用判断该低位字节的值是否还是非零，来区分接收到的是功能控制键还是ASCII字符键的。请注意上述例子中的具体用法。

10h的子功能码为3的中断，完成的是把光标当前位置读到dx寄存器中，高位字节放的是行号，可取值0~24，低位字节放的是列号，可取值0~79，但我们习惯上更愿意用1~25和1~80来表示光标的行列值，为此，在程序中对读来的行列值都执行了加1操作。在读定位光标的过程时，请注意把dx中的值传入PASCAL程序中的变量row和COLUMN中的具体办法。我们在前边已详细叙述过这一问题。

此外，在汇编语言子程序中，各种定义说明的次序是很重要的，例如ASSUME伪操作就必须出现在PUBLIC说明之前。

有了上述说明，读懂给出的程序实例，读者就有能力试着设计自己的实用程序。

9.4 实现音响控制和仿真电子琴的程序

PASCAL程序调用汇编语言子程序的第二个例子，是通过接口地址完成入/出接口上的读写功能。该程序的功能可实现音响控制，这要用到微机系统中的小喇叭（SPEAKER）和计时器（TIMER）两个设备。小喇叭的接口地址为#61（#代表16进制，相当于汇编语言中的61h），计时器的接口地址为#42。计时器用于决定音响的频率，再控制小喇叭的接通与关闭，就可以按我们的意图产生期望的音响，#4f和#4d是用于接通和关闭小喇叭的控制码。

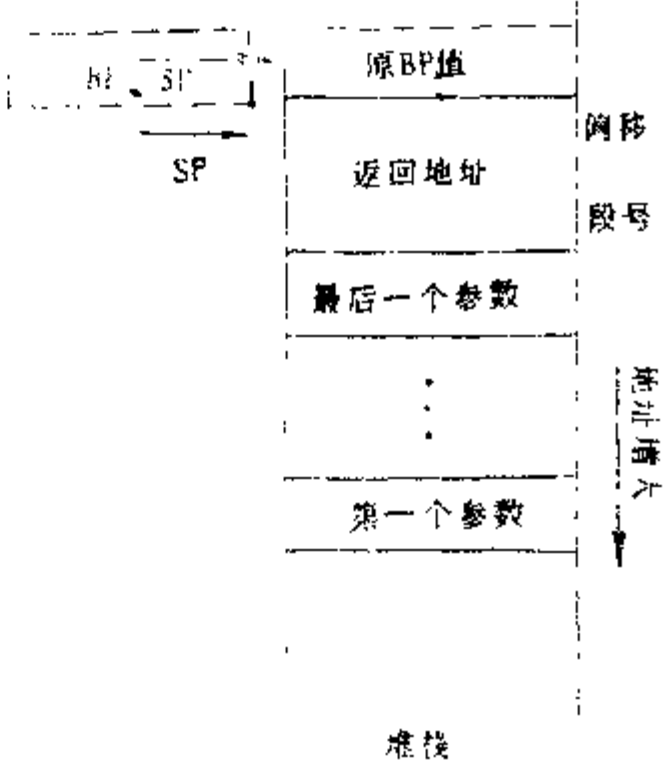
这个程序中的关键技术，是向接口写入数据的操作，这是在如下一个叫做PORTOUT的汇编语言子程序中实现的，它是PASCAL程序的一个外部过程，用到接口地址和所写数据两个参数。这个办法不仅可用来控制向小喇叭发送数据（接通和关闭的命令码），向计时器发送数据（发音频率），也是向各种接口发送数据的通用方式。

下面是该PASCAL程序和汇编子程序的源程序清单。

```

program blaiser-blact (input, output);
  const
    speaker = #61;

```



```

        timer          = # 1;
        toggle-on      = # 1;
        toggle-off     = # 1;
        max-count      = 250;
        scaler          = 3;
    var
        code, count, i: word;

    procedure portout (port:addr:word; data:byte); extern;

begin
    repeat
        write ('press enter to fire') ;
        readln;
        portout (speaker, toggle-on) ;           ! 接通喇叭

        for count:= 0 to max-count do
            [for i:=1 to 2000 do code:=code;      ! 延迟一段时间
             code:=sqr (count) div scaler;
             portout (timer, lobyte (code)) ;      ! 送决定频率的数据
             portout (timer, hibyte (code)) ;      ! 送决定频率的数据
            ] ;

            portout (speaker, toggle-off)          ! 断开喇叭
        until count= # ffff;
    end.

    coder    segment
        assume cs:coder
        public portout
    portout  proc      far
        push    bp
        mov     bp,    sp
        mov     dx,    [bp+ 8]
        mov     al,    [bp+ 6]
        out     dx,    al
        pop     bp
        ret     4
    portout  endp
    coder    ends
end

```

这个程序的功能，是产生一个类似于爆炸的音响效果。接通喇叭后，用for 语句给出一个决定音响频率的数据，其变化规律为0~250的平方值除以2。整个程序是个死循环，每按一次回车键，发声一次，发声过程一结束，断开喇叭，以停止发声。整个程序的结束，可以用键入CTRL/C实现。

我们可以在这个程序的基础上，稍加改动，实现一个简单的模拟电子琴的程序。其控制发声的办法不变，再加上控制音阶频率和发音时间的手段就可以了。控制音阶频率，用的是常量数组的八个元素分别去除# 120000，得到八个音阶频率。用键入终端键盘上的数字键1~8控制选用八个音阶中的对应音阶。控制发音时间的办法是，键入数字键9 将关闭喇叭，停止发音，连续键入的两个1~8的数字键不相同，也将关闭喇叭，并且在再次接通喇叭后，按新的音阶发声。

程序的第一个语句用于初始化地址为# 42 的计时器。程序的核心部分，是用一个repeat 语句控制的。先用单字符输入方式，读来一个字符，当本次输入的字符与上一次不同时，关掉喇叭。接下来检查输入的字符是否为'1'..'8'，若是，则选中一个音阶并控制发声；否则，为字符'9'，关闭喇叭，不为字符'9'，则认为输入的是非法字符，响铃提示出错。键入CTRL/C时，程序结束。

下面是该程序的源清单。

```

program piano (input, output) ;
type arr=array[1..8] of integer4;
const ar=arr (262, 294, 330, 347, 392, 440, 494, 524) ;
    speaker = # 61;
    timer    = # 42;
    constant = # 120000;
var i, il:byte; w:word; j:integer4;

function dosxqq (a, b:word) :byte; extern;
procedure portout (port_addr :word; data:byte) ; extern;
begin
    portout (# 43, 2 # 10110110) ; il:= 0 ;
    repeat
        repeat i:=dosxqq (6, 255) ;
            if (i< >il) and (i< > 0) then portout (speaker, # 4d)
        until i< > 0 ;
        if (i>=49) and (i<=56)
            then [j:=constant div ar[ord (i-48) ] ;
                w:= wrd (j) ;
                portout (speaker, # 4f);
                portout (timer, lobyte (w) ) ;
                portout (timer, hibyte (w) ) ;
                for w:= 0 to 2000 do i:=i ;
                il:=i]
    
```

```

        else if i=57 then portout (speaker, #4d)
            else write (chr (07))
        until i=3;
        portout (speaker, #4d)
    end.

```

参照上述办法，我们也可以很容易地实现一个从入/出接口读来相应数据或接口工作状态码的汇编子程序，并在 PASCAL 程序中将其作为外部函数来引用。此时，要在 PASCAL 程序中作如下说明：

```

FUNCTION PORTIN (PORT-ADDR:WORD):BYTE;
EXTERNAL;

```

下面是 PORT-IN 汇编子程序的源程序清单。

```

coder      segment byte public
            assume cs:coder
portin      proc    far
            push    bp
            mov     bp, sp
            mov     dx, [bp+6]
            in      al, dx
            pop     bp
            ret     2
portin      endp
coder      ends
            end

```

提请读者注意，微机中很多入/出接口都是以字节方式传送的，入/出操作中必须遵守有关规定。

9.5 协同运行多个程序和使用公用数据区

协同运行多个程序和使用公用数据区的最简便办法，是用一个汇编语言的程序作为总控程序，在它里边开出公用数据区，并把用到的多个程序作为子进程 (SUBPROCESS) 调用，再通过传送或得到公用数据区的起始地址的办法使用一片或多片公用数据区，以满足快速进程通讯的要求。许多刚接触程序设计的人员，不大体会此种要求的必要性。作为结论，可以告诉大家，要实现大型的实际事务或业务处理，是很难用一个很大的程序全面完成的，更合理的方案，是让很多个执行程序协同运行、适当通讯，共同完成全部的处理功能。作为这种用法的一个实例，我们给出如下的一个汇编语言的程序。它的功能是：

(1) 开辟 10000 个字节 (初值均填小写字母 a) 的一片内存区，供该汇编语言程序 和它调用的另外两个 PASCAL 程序共同使用，以表明多个程序共同使用同一片内存 (公用数据区) 的方法和效果。

(2) 在该汇编程序中，可以通过功能清单 (MENU) 选择运行用户的两个 PASCAL 程序或系统提供的汇编程序 (ASSEMBLER)，或结束本程序运行，退回到操作系统状

态。两个 PASCAL 的源程序清单附在汇编语言程序的后边。第一个用于在终端屏幕上显示九九乘法表,之后再显示公用数据区的前100个单元的内容并清其为空格。第二个用于对指定的文件建立一个分页打印的 TEXT 文件,之后再把公用数据区的前100个单元的内容变为 '@' 到 'Y' 这26个字符组成的序列的重复串。汇编程序用于对指定的汇编语言的程序进行汇编处理,与在操作系统控制下调用汇编程序用法类似。

(3) 汇编程序调用了每个可执行程序之后,被调用的程序(子进程)将投入运行,当其运行完毕之后,控制将返回到汇编程序(父进程)中,而不是返回到操作系统状态。这在一个程序中反复地选择运行多个执行程序所必须的,是协同运行多个程序并使用指定的公用数据区所必须的,与通过批处理语言执行多个程序有明显区别。

(4) 前边已说过,汇编总控程序开始时,已设置公用数据区的每个存储单元为 'a' 字符,第一个 PASCAL 程序将读并显示公用数据区前100个单元的内容,还要清其为空格,而第二个程序将向公用数据区写入新的内容。作为检查运行效果的手段,我们可以用不同的次序调用两个 PASCAL 程序,在屏幕上将得到不同的显示内容。如果先运行第一个 PASCAL 程序,屏幕上将得到100个 'a' 字母(汇编程序设置的)。如果先运行第二个 PASCAL 程序,再运行第一个 PASCAL 程序,屏幕上将得到第二个 PASCAL 程序写进公用数据区的内容,若接下来再运行第一个 PASCAL 程序,屏幕上将显示100个空格(第一个 PASCAL 程序前一次运行时设置的)。

程序清单之后的运行结果,就是在总控程序运行后,按先运行第一个 PASCAL 程序,再运行第二个 PASCAL 程序,并再次运行第一个 PASCAL 程序的次序,以及在屏幕上得到的显示结果。

当然,我们也可以在此调用汇编程序。最后通过选定最后一项功能清单,结束整个程序的运行过程。

下面给出三个程序的源程序清单和运行结果。

```

loop    MACRO  NHZ,          ; 提示声音, XZU
        mov     ah,          0
        mov     dl,          0
        int     21h
        ENDM

men_sel MACRO  func
        lea     cx, func
        mov     ah, 0
        int     21h
        ENDM

DSEG SEGMENT PUBLIC
COMAREA DB 10000 dup('a')    ; 设置公用数据区
COMMAND1 DB 'listgg.exe', 0H
COMMAND2 DB 'page.exe', 0H
command3 db 'MENU.COM', 0h
command  db 0                ; 设置程序命令文件为空

```



```

men-sel msg
men-sel titl1

ITEM:  MOV     CX, 4
      MOV     SI, OFFSET CAS2
      MOV     DAT2, OFFSET CAS1
      SUB     SI, DAT2

LOP2:  PUSH    CX
      MOV     DX, DAT2
      ADD     DAT2, SI
      mov     ah, 9
      int     21h
      POP     CX
      LOOP    LOP2

      sub     dat2, si
SELECT: XOR     DI, DI
cl:     beep    2
      MOV     DX, DAT2
      mov     ah, 9
      int     21h

      CMP     DAT2, OFFSET CAS4
      JNE     NOEND
      MOV     DAT2, OFFSET CAS1
      xor     di, di
      jmp     ag1
NOEND:  ADD     DAT2, SI
ag1:

      men-sel cls1
      MOV     DX, DAT2
      mov     ah, 9
      int     21h
      men-sel cls0
again:  MOV     AH, 6
      mov     dl, offh
      int     21h
      cmp     al, 0
      jz      again

```

```

        cmp     al, 20h        , space bar
        Jne     return
        inc     di
        JMP     FAR PTR CI

return:  CMP     AL, 13
        pushf
        beep
        popf
        JE      OK2
        JMP     again
OK2:     men-sel  class
        men-sel  MSG1
        RET
MENU    ENDP

STAR    PROC    FAR
        MOV     AX, DSEG
        MOV     DS, AX
        push    ds
        mov     ax, 0          , 送公用数据区地址入 0:180h
        mov     ds, ax
        mov     bx, 180h
        mov     [bx], dseg
        add     bx, 2
        mov     ax, offset comarea
        mov     [bx], ax
        pop     ds

        mov     ah, 4ah        , ES中已是数据段的基地址
        mov     bx, 400h
        int     21h            , 申请内存, 大小为400h

begin:   men-sel  class
        men-sel  cls0
        CALL    MENU

        cmp     di, 3
        jNE     exec0
        JMP     FINISH

```



```

        if i mod 25 = 0 then writeln;
        write ('input any character to continue! ');
        read (ch);
    end;

program pages (input, output);
var
    name, name1 : string[120];
    fi, fo : text;
    a, i, j, k, p : integer;
    ch : char;
    xun : ads of words;
    ad : ads of char;
begin
    write ('number of line per page? '); readln (i);
    write ('number of copy per page? '); readln (k);
    write ('input file name? '); readln (name);
    write ('output file name? '); readln (name1);
    assign (fi, name); reset (fi);
    assign (fo, name1); rewrite (fo); a := 0;

    while not eof (fi) do
        [a := a + 1; i := 0;
        writeln (fo, '':30, name, '':10, a); writeln (fo);
        repeat
            while not eof (fi) do
                [ch := fi^; write (fo, ch); get (fi);
                readln (fi); writeln (fo);
                j := j + 1;
            until (i = j) or eof (fi);
            for p := 1 to k - 1 do writeln (fo);
        ];
        close (fi); close (fo);
        write ('.....use comarca.....');
        xun.s := 0; xun.r := 16 # 180; ad.s := xun^;
        xun.r := xun.r + 2; ad.r := xun^;
        for p := 1 to 100 do
            [ad^ := chr (p mod 25 + 65); ad.r := ad.r + 1;
        end
    end

```

运 行 结 果

乘 法 九 九 表

	1	2	3	4	5	6	7	8	9
1	1								
2	2	4							
3	3	6	9						
4	4	8	12	16					
5	5	10	15	20	25				
6	6	12	18	24	30	36			
7	7	14	21	28	35	42	49		
8	8	16	24	32	40	48	56	64	
9	9	18	27	36	45	54	63	72	81

--- use comarea

---output comarea data--- --

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

input any character to continue!

number of line per page ? 46

number of empty per page ? 4

input file name ? wap.asm

output file name ? wap.pr

use comarea

在上面给出的例子中，两个PASCAL程序容易看懂，而对总控程序和使用公用数据区的问题，尚需进行某些说明。

总控程序，对不熟悉汇编语言的读者，可能显得长了一点，但其中大部分的内容，是用于显示和选择功能清单，不是我们关心的重点，只有从标号STAR开始的最后一部分，才是该程序的核心内容，指令不多，应该说是容易读懂的。

从标号BEGIN开始之前的三行到程序结束，是实现在程序之内调用执行另外的程序的指令。这里的关键技术，是正确实现中断21h的子功能码为4ah和4bh的两个功能调用。

4ah功能调用，完成修改内存分配情况。

开始前，功能码4ah放在AX寄存器的高位字节AH中。以节(paragraph)为单位的新申请的内存区的大小放在BX寄存器中，修改块的段号放在ES寄存器中。

调用后，若处理机进位标志被清，表示修改成功，BX中给出可用的最大内存块的大小。若修改失败，错误码放在AX寄存器中。

错误码为7，表明内存控制块被破坏；

错误码为 8，表明可用内存不够；

错误码为 9，表明ES中给的段号不正确。

4bh 功能调用，完成把另一个执行程序装入内存并使其投入运行的操作。当被调入的程序结束运行过程后，控制将返回主调用程序，在需要时，还可以接收被调用程序的一个返回码。调用4bh之前，必须先调用4ah，为被调入的程序分得必要的内存空间。

开始前，功能码4bh放在AH中，附加的子功能码（00表示装入并执行一个程序，03表示装入但不建立程序参数块，也不执行装入的程序）放在AL中。程序参数块的段号和位移分别放在ES和BX寄存器中。程序的说明段和位移放在DS和DX寄存器中。

返回时，进位标志为零，表示调用成功，并且破坏除CS和IP之外的全部寄存器，包括堆栈指针在内。若调用失败，错误码放在AX寄存器中。

错误码为1，表明子功能码非法；

错误码为2，表明指定的程序文件找不到；

错误码为5，表明拒绝存取；

错误码为8，表明分配的内存空间不够；

错误码为0Ah，表明环境无效；

错误码为0Bh，表明参数块的格式不对。

程序参数块，对0子功能码有如下格式：

WORD 环境块的段指针

DWORD 在80h中的命令行的指针

DWORD 在5ch中的被传送的默认的FCB的指针

DWORD 在6ch中的被传送的默认的FCB的指针

下面我们就看一看上述两种功能调用在程序中的具体实现。

程序的前面部分，给出的是数据段的内容。其中与上述两种调用有关的是：

- 三个可执行程序的文件名，标号分别为COMMAND1, COMMAND2和COMMAND3；三个用于存放三个文件名地址的内存字，标号为acom1, acom2和acom3。

- 程序参数块用的7个字，标号为PARAMETER。

在程序中ASSUME伪命令中，已指明DS:DSEG, ES:DSEG，即让数据段基址寄存器和外部段基址寄存器都保存该程序的数据段的基址。这样，程序的环境段和参数块段的段地址就可以用DS和ES给出。偏移值很容易用OFFSET取得。

显示程序的功能清单和进行功能选择，都是在MENU子程序中完成的。在那里，按空格键可以选择功能，按回车键开始执行选中的功能。选中的功能，是用di寄存器中的0~3这四个值表示的，分别表示选中四个功能中的哪一个，通过这个di值，就可以计算三个执行程序的文件名的地址，或控制转到结束程序运行过程的指令之处。

说明至此，我们想读者是可以看懂在一个程序之内，是怎样完成调用另外的程序的具体方案了。

下面讨论使用公用数据区的实现方法。在总控程序中，开了10000个字节的一个数据区，其段号由DS给出，偏移值由COMAREA给出。在程序开始处，把这两个值分别存到内存绝对地址180h和182h中。目的是为了让被调用的其它程序能找到这片内存区。

在被调用的两个PASCAL程序中，首先要通过地址变量找到保存在内存绝对地址十六进制的180和182中的这片公用内存区的段号和偏移值，接下来，通过地址就能访问这片

公用内存区。为此，在PASCAL程序中说明了一个字的地址变量XUN和一个字符的地址变量ad。首先让XUN.s为0，XUN.r为#180，取出XUN^的值并赋给ad.s，使ad的段号为公用数据区的段号，然后使XUN.r为#182，取出XUN^的值并赋给ad.r，使ad的偏移地址为公用数据区的偏移地址，这样，ad对应的正好是公用数据区的头一个单元。这之后，可以通过修改ad.r的值来访问公用数据区的任何一个单元。这些问题可以在PASCAL程序中看得很清楚。

上面这个例子是协同运行多个程序和使用公用数据区的典型用法，程序本身虽无实际应用意义，但它给出的原则和具体实现方法，却有极好的实用参考价值，是一个大的实用程序系统的基础骨架。认真读懂这个例子，必定对提高实际程序设计的能力有所裨益。

在PASCAL程序中，也能完成调用另外的程序的功能。由于PASCAL编译程序没直接提供前面讲的功能码为4ah和4bh的中断21h的调用能力，因此，要在这个PASCAL程序中，说明和使用两个用汇编语言实现的外部过程FREEMEM和EXECUTE，分别用于改变内存分配情况和执行指定的程序。在PASCAL程序中调用汇编语言子程序的有关技术已在上一节详细介绍过，此处不再说明。我们将简单说明PASCAL程序和两个外部过程中的几个问题。

下面是三个源文件的清单。

```

program multipro (input, output) ,
const
    esc=chr (27) ;
type aad=ads of lstring (10) ;
var
    command:array [1..3] of lstring (16) ;
    cesxqq [extern] :word; 1
    adrs:aad;
    fln:lstring (20) ;
    t1:byte;
    t2:word;
    sel, count:integer;
value
    command[1]:='list99.exe' ,
    command[2]:='page.exe' ,
    command[3]:='asm.exe' ;

procedure freemem (i:word) ;          extern; 1 修改内存分配情况
procedure execute (i:word) ;          extern; 1 执行一个指定的程序
function dosxqq (i, j:word) :byte;    extern; 1 单个字符的入/出

procedure clss;
begin write (esc, '~2J'); end;        1 清屏
procedure cls0;                        1 恢复正常显示属性

```

```

begin write (esc, '[0m') end;
procedure cls;
begin write (esc, '[1; 45m') end;      ! 置规定的显示属性
procedure beep;                          ! 响铃
begin write (chr (7) ) end;
procedure men-sel (func, attr:integer) ; ! 按指定属性显示
begin write (esc, '[', attr:l, 'm') ;   ! 一行功能清单信息
case func of
0:write (esc, '[11; 14H 1- 9 * 9 table      ');
1:write (esc, '[12; 14H 2- 9 * 9 matrix      ');
2:write (esc, '[13; 14H 3- assembler        ');
3:write (esc, '[14; 14H 4- EXIT TO SYSTEM ');
end; cls;
end;

begin (* main *)
t2:=cesxqq;
freemem (t2) ; count:=0;
repeat beep;
write (esc, '[2J', esc, '[0m') ;
write (esc, '[5; 10H      Program Name') ;
write (esc, '[6; 10H      Tsinghua 88.7') ;
write (esc, '[7; 10H      - - - - -      ');
write (esc, '[8; 10H      >>> mein menu <<<      ');
write (esc, '[9; 10H      .                ');
write (esc, '[20; 56H', esc, '[0;7m      ', esc, '[0m') ;
write (esc, '[21; 8H', esc, '[36m Press', esc, '[0; 7m');
write (' Spece Ber      ', esc, '[0; 36m select item; Press');
write (esc, '[0; 7m Return      ', esc, '[0; 36m exec', esc,
'[0m');
for sel:= 0 to 3 do men-sel (sel, 0) ;
repeat
men-sel (count,7) ;
repeat t1:=dosxqq (6,255) until t1<>0;
if t1=#20
then[men-sel (count, 0) ;
count:=(count+1) mod 4]
else beep
until t1=13;
if count<3 then

```



```

        [class beep;
          adrs:=ads command[count+1];
          t2:=adrs.r;
          execute (t2) ]
until count=3;
class
end.

codesg segment
        assume cs:codesg, ds:codesg
        public freemem
freemem proc far
        PUSH BP
        PUSH ES
        MOV BP, SP
        mov ax, word ptr [bp+8]
        MOV FS, ax
        mov bx, 2000h
        mov cx, 4a00h
        int 21h
        POP ES
        POP BP
        ret 2
freemem endp
codesg ends
end

codesg segment
zhan dw 2 dup (?)
fln db 30 dup (0)
parameter dw?
dw 6 dup (?)
con: db?

        assume cs:codesg, ds:codesg, es:codesg
execute proc far
        public execute
start: push ds ; 保存原ds, es和bp寄存器
        push es ; 存器的值到堆栈中
        push bp
        mov bp, sp
        mov i, [bp+10] ; 取来文件名的偏移地址

```

```

        mov     ax,     codeseg
        mov     es,     ax
        mov     di,     offset fln
        mov     ci,     byte ptr [di]
        sub     ch,     ch
        add     si,     1
        cld
        repnz   movsb                    ; 把文件名传入fln!x
        mov     ax,     codeseg
        mov     ds,     ax
        add     di,     1
        mov     byte ptr [di], 0

        mov     ax,     codeseg
        mov     ds,     ax
        mov     es,     ax

        mov     bx,     offset zhan
        mov     word ptr [bx], ss        ; 保存ss和sp的值
        mov     word ptr [bx+2], sp
        mov     bx,     offset parameter
        mov     dx,     offset fln
        mov     ax,     4b00h
        int     21h                      ; 调用被执行的程序
        mov     ax,     cs
        mov     ds,     ax
        mov     bx,     offset zhan
        mov     ss,     word ptr [bx]    ; 恢复ss和sp的值
        mov     sp,     word ptr [bx+2]
        pop     bp                        ; 恢复PASCAL程序的
        pop     es                        ; 原bp, es和ds的值
        pop     ds
        ret     2

execute endp
codesg ends
end

```

在这个例子中，该PASCAL程序的功能，与前面讲过的汇编语言的总控程序的功能是类似的，只是少了给出公用数据区的功能。前两个语句，实现修改内存分配的功能。以下的多个WRITE语句和接下来的那个FOR语句用于显示功能清单（包括操作提示信息），接下来的REPEAT语句，完成选择功能，加亮显示所选功能的一行信息，按空格键，变

换选择的功能项，按回车键将结束 REPEAT 语句。按其它键均属非法，响铃后等待输入新的字符。变量 COUNT 用于记录选中的功能项的编号（从 0 到 3）。在程序最后的几行，将按 COUNT+1 的值，找出所选功能对应的执行程序的文件名，并调用该程序使其投入运行。当选择了退到操作系统的功能项时，该 PASCAL 程序将结束运行，整个程序是用最外层的一个 REPEAT 语句控制的。看懂这个 PASCAL 程序，比读懂前一个汇编程序要容易得多。

两个外部过程中，没什么新内容。调用 FREEMEM 过程中的实际参数是 PASCAL 程序的 ES 寄存器中的值，是用 PASCAL 程序预定义的一个符号 CESXQQ 给出的。

调用 EXECUTE 过程用的实际参数是被调用程序的文件名的偏移地址。在这个过程中，首先要保留主调用者（PASCAL 程序）程序的 ds, ES 和 BP 的值进堆栈，通过参数把被调用程序的文件名传入本过程中的 fln 区，文件名之后的第一个字节写入 0 值。再保留堆栈指针到本过程中指定的单元，这里用的是 ZIHAN 标号区。之后就通过调用功能码为 4bh 的中断 21h，使被调用程序装入内存并启动执行。被调用程序执行结束后，要首先恢复这个过程的堆栈指针的段号和偏移值，再恢复这个过程的主调用者程序的 bp, es 和 ds 值，并结束本过程，返回主调用者程序。

有能力又有兴趣的读者，可以比较前面两个例子程序，看一看 PASCAL 程序和汇编语言程序的同异之处，加深对两个例子的理解。

9.6 在程序中查询系统信息

在某些程序运行的过程中，有时需要了解所用的微机系统的硬、软件配置等系统信息。例如一些大的通用软件的安装（INSTALLATION）程序，它必须首先了解系统的硬软件配置情况，如配了几个磁盘驱动器，都是什么型号的，所用终端的分辨率，彩色还是单色，打印机的特性，有无安装浮点运算的选件，所用操作系统的版本号，以及系统中配备的内存的容量等等，然后按系统实际配置情况安装或生成一个实用软件。查询这些系统信息的功能，需要调用系统的中断功能调用，必须使用一个汇编语言实现的外部过程。

先介绍与系统信息有关的几个数据结构。

一、系统配置表

系统配置表是由 10 个内存字组成的：

第一个字：保存 BIOS 软件的版本号

第二个字：保存内存容量，单位是 paragraph，通称为节，每节由 16 个字节组成

第三个字、第四个字：保留

第五个字：可用终端屏幕页数，与保存屏幕信息的显示器的内存区容量有关

第六个字：屏幕信息块地址指针

第七个字：系统中配置的硬软盘驱动器数目

第八个字：0 号软盘驱动器控制块指针

第九个字：1 号软盘驱动器控制块指针

第十个字：温盘控制块指针（重复 0 到多次）

二、屏幕信息控制块

屏幕信息控制块由10个字节组成:

第一个字节: 状态信息, 包括活动/类型/插件槽等信息

第二个字节: 扫描寄存器偏移值

第三、四个字节: 作为字, 给出数据缓冲区的段地址

第五个字节: 颜色信息, 高四位, 字符颜色

低四位, 背景颜色

第六个字节: 光标所在的行号 (0...25)

第七个字节: 光标所在的列号 (0...79)

第八个字节: 显示属性

第九个字节: 附加方式字节 (确定板的中断状态)

第十个字节: 第二个附加方式字节

三、磁盘驱动器控制块

磁盘驱动器控制块由5个字节加2个字组成:

第一个字节: 磁盘类型

第二个字节: 驱动器状态

第三个字节: 每个扇区的字节数

第四个字节: 每个磁道的扇区数

第五个字节: 每个柱面的头数

第一个字: 每个磁盘驱动器的柱面数

第二个字: 每个磁盘驱动器的扇区数

下面说明查询这些表的方法。对IBM PC和WANG PC来说, 二者不完全相同, 二个厂家的微机不完全兼容, 其中很重要的一条是访问BIOS (基本输入输出软件系统) 的方法不一样。BIOS被看作为微机系统中最靠近硬件的软件, 是DOS操作系统的基础软件。直接用BIOS功能的程序难以移植, 许多所谓的IBM PC的兼容机, 有可能在这些地方不兼容。BIOS的功能是通过20h以下的中断调用实现的。而在WANG PC中, 是通过INT 88h加一个功能码给出部分 (而不是全部) 调用功能的。从这个意义上看, IBM PC的软件开放性更好些。

从查询系统信息的角度看, WANG PC机的子功能码为01h的中断88h更有用, 它将把系统配置表的地址取到ES:BX寄存器中。有了系统配置表, 就能找到其它有用的系统信息。为此, 可以在PASCAL程序中说明如下一个外部过程:

```
PROCEDURE INF (VAR SYSCONF:ADS OF WORD); EXTERN;  
用变量参数SYSCONF接收系统配置表的地址。
```

下边是PASCAL程序和它的外部过程的源清单。程序的功能是取来上述的系统信息, 并把其内容显示在终端屏幕上。

(* 查找系统设备表并输出屏幕信息块内容 *)

```

program sysconf(input, output);
type
  adw=ads of word;
  table=array [0..9] of word;
      (* 系统设备表的内容
      version, memsize, reserved1, reserved2, serment,
      sernptr, diskent, floppy0, floppy1, winstat *)

  serninf=record
      state:      byte;
      sernoff:    byte;
      bufseq:     word;
      color:      byte;
      row:        byte;
      col:        byte;
      attr:       byte;
      auxmod:     byte;
      auxmd2:     byte;
  end;
  diskinf=record
      disktyp      [0] :byte;      ! 盘类型
      diskstat     [1] :byte;      ! 驱动器状态
      byte-sector  [2] :byte;      ! 字节/块
      blk-trace    [3] :byte;      ! 块/磁道
      head-cylind  [4] :byte;      ! 头数/柱面
      cylind-disk  [5] :word;      ! 柱面数/盘
      sector-disk  [7] :word;      ! 扇区数/盘
  end;
var ad:adw; i:integer; b:byte;
    ass, arr, w:word;
    adsstable: ads of table;
    adsernblk: ads of serninf;
    adsdriver: ads of diskinf;

procedure sconfig (var as, ar:word); extern;

begin
  sconfig (ass, arr);
  ad.s:=ass; ad.r:=arr; w:=ad.r;
  adsstable.s:=ad.s; adsstable.r:=ad.r;
  adsernblk.s:=ad.s; ad.r:=ad.r+10; adsernblk.r:=ad^;

```

```

adsdriver.s:=ad.s; ad.r:=ad.r+adsdriver.r+ad^;

writeln ('系统设备表地址: ', ad.s:4:16, ' ', w:4:16);
writeln ('系统设备表的内容: '); sour:=w;
for i:=0 to 9 do writaln ('':8, dsctable^ [i] :4:16);
writeln;

writeln ('屏幕信息块偏移地址: ', adscrnblk.r:4:16);
writeln ('屏幕信息块的内容: ');
with adscrnblk^do
  [writeln ('':10, state :4:16, ' ', scno:4:16);
   writeln ('':10, buffered :4:16, ' ', colors:4:16);
   writeln ('':10, row :4:16, ' ', col :4:16);
   writeln ('':10, attr :4:16);
   writeln ('':10, auxmod :4:16, ' ', auxmd2:4:16) ];
writeln;

writeln ('软盘驱动器 (0) 控制块偏移地址: ', adsdriver.r:4:16);
writeln ('软盘驱动器 (0) 控制块的内容: ');
with adsdriver^do
  [writeln ('':10, disktyp :4:16, ' ', diskstat :4:16);
   writeln ('':10, byte-sector :4:16, ' ', blk-trace :4:16);
   writeln ('':10, head-cylind:4:16, ' ', cyclind-disk:4:16);
   writeln ('':10, sector-disk :4:16) ];

```

end.

程序的运行结果如下所示:

系统设备表地址: 0544 0099

系统设备表的内容:

```

0119
8000
0003
0000
0001
434E
0005
2220
2234
2248

```

屏幕信息块偏移地址: 434E

屏幕信息块的内容:

00D2 0000

0000 00C3

0018 0011

0000

0044 0000

软盘驱动器 (0) 控制块偏移地址: 2220

软盘驱动器 (0) 控制块的内容:

0000 0085

0002 0009

0002 0050

05A0

用汇编语言实现的外部过程的源清单:

```
codesg    segment public 'code'
            assume  cs:codesg
            public  sconfig
sconfig    proc      far
            push    bp
            push    ds
            push    es
            mov     bp,                sp
            mov     al,                1
            int     88h
            mov     si,                [bp+12]
            mov     ds: [si],          es
            mov     si,                [bp+10]
            mov     ds: [si],          bx
            pop     es
            pop     ds
            pop     bp
            ret     4
sconfig    endp
codesg     ends
end
```

在IBM PC机系统中, 查询系统信息是用INT 21h, INT 11h和INT 12h完成的。

21h的功能码为30h的中断调用, 将把DOS的版本号取到ax寄存器中, 其中低位字节是版本号中圆点左面的版本号, 高位字节中是圆点右边的版本号, 如AX中为1E03h, 表明当前DOS的版本号为3.30。

12h中断调用, 将把内存总容量取到ax寄存器中, 单位是1KB。

11h中断调用, 将把与系统设备配置有关的信息取到ax寄存器中。该信息是以一个字的形式给出的, 用字中不同的二进制位分别表示有关信息, 具体规定是,

(AX) 15,14: 连接的打印机数目;

(AX) 12 : 连接的游戏I/O;

(AX) 12 : 连接的RS232的数目;

(AX) 7,6 : 驱动器数目;

(AX) 5,4 : 初始显示模式: 00—不用;

01—40×25黑白 (彩色板)

10—80×25黑白 (彩色板)

11—80×25黑白 (黑板)

(AX) 3,2 : 底板RAM容量 (00=16K, 01=32K, 10=48K, 11=64K)

(AX) 0 : 系统有无软盘驱动器

(AX) 1,8,13不用

PASCAL程序的源清单:

```
program test (input,output);
```

```
const esc=chr (27);
```

```
var device-word,bits:word;
```

```
procedure get-sysdev (var w:word); extern;
```

```
procedure get-dos-ver (var w:word); extern;
```

```
procedure get-memsize (var w:word); extern;
```

```
begin
```

```
write (esc,' [2J'); (*clear the screen*)
```

```
get-sysdev (device-word);
```

```
write (esc,' [4; 8H','device-word:', device-word:4:16);
```

```
      (*examine all bits in the word*)
```

```
write (esc,' [6; 3H floppy driver:');
```

```
if device-word and #01=1
```

```
    then write ('Have') else write ('No');
```

```
write (esc,' [7; 3H board RAMsize:');
```

```
case device-word and #0c of
```

```
    #00: write ('1 * 16k');
```

```
    #04: write ('2 * 16k');
```

```
    #08: write ('3 * 16k');
```

```
    #0c: write ('4 * 16k')
```

```
end;
```



```

write(esc, '[8;3H primit module:') ;
case device-word and #030 of
    #10: write('40*25 white-black (color board) .') ;
    #20: write('80*25 white-black (color board) .') ;
    #30: write('80*25 white-black (no-color board).') ;
end;

write(esc, '[9; 3H driver number:') ;
write(device-word and #0c0 div 32:3) ;

write(asc, '[10; 3H rs232 number:') ;
write(device-word AND #0c00 div 312:3) ;

write(esc, '[11; 3H printer number:') ;
write(device-word AND #c000 div 16384:3) ;

write(esc, '[12; 3H DOS version:') ;
get-dos,ver(bits) ;
write(lobyte(bits):1, '.', hbyte(bits):2);

write(esc, '[13; 3H total RAMsize:') ;
get-memsize(bits) ; write(bits:5, '*1k bytes')
end.

```

程序的运行结果给出如下:

```

device-word      :526F
floppy   driver   :Have
board    RAMsize  :4*16k
primit   module   :80*25 white-black (color board)
driver   number   :2
rs232    number   :0
printer  number   :1
DOS      version  :3.30
total    RAMsize  :640*1k bytes

```

用汇编语言实现的三个外部过程的源清单:

```

codesg segment public 'code'
    assume cs:codesg

    public get-sysdev
get-sysdev proc far
    push bp
    mov bp, sp
    mov bx, [bp+6]

```

```

        int     11h
        mov     [bx] , ax
        pop     bp
        ret     2
get-sysdev endp

        public  get-dos-ver
get-dos-ver proc far
        push    bp
        mov     bp, sp
        mov     ah, 30h
        int     21h
        mov     bx, [bp+6]
        mov     ds:[bx] , ax
        pop     bp
        ret     2
get-dos-ver endp

```

```

        public  get-memsize
get-memsize proc far
        push    bp
        mov     bp, sp
        int     12h
        mov     bx, [bp+6]
        mov     [bx] , ax
        pop     bp
        ret     2
get-memsize endp
codesg ends
        END

```

第十章 文件和磁盘的应用技术

合理的管理、使用文件与磁盘，是改进应用软件质量，完善微机系统应用环境的重要措施。在前面各章中，我们仅简单使用 PASCAL 语言直接提供的文件操作的功能。在一些实际应用中，常常感到对某些具体问题没有办法用这些功能加以解决。比较突出的一个问题，是没有办法得到与使用一个通用的、读写任意文件的一个记录的过程。这里说的任意一个文件，是指不同文件的记录的组成情况和记录长度可随意变化，通用的读写过程是指能用于读写上述任何一个文件的过程。按以前学过的办法，我们只能分别说明每一个不同的文件，并分别打开，各自单独操作。在对磁盘的使用上，我们只用 PASCAL 语句操作文件，并不直接操作磁盘。我们将在本章，向读者介绍对文件和磁盘的另外一些用法，进一步提高程序设计的能力。

10.1 文件控制块 (FCB) 和文件用法

先看一个示意性的小程序。该程序将读进 4 个不同记录长度的字符文件的内容，并把它们显示在终端屏幕上。

```
program FILUSE (input, output);
type
    a=string (200);
var
    f      : file    of string (1);
    d1,d2  :         a      ;
    fads   : ads    of fcbfqq;
    brr    : ads    of adrmem;
    datar  : adr    of      a;
    wr     : ads    of word;
    i, j, size, rn : word;
    fnl    : text;

    procedure open ;
    begin
        readln (fnl, f);
        readln (size);
        f.mode:=direct; i.trap:=true;

        fads:=ads f      ;      (* write file record size      *)
        f.s:=fads.s      ;      (* into fcb's proper location *)
        f.r:=fads.r+10;
```

```

    wr^ := size;

    brr.s := fads.s ,           (* write file buffer'address *)
    brr.r := fads.r+24;        (* into fcb'proper location *)
    datar := adr dl ,
    brr^ .r := datar.r
end;
begin                          (* main program using to *)
    assign (fml,'namlenth.dat') , (* test open procedure validity *)
    reset (fml) ;
    for i:=1 to 4 do
        [open; reset (f) ;      (* open the file f form *)
        if ord (f.errs) < > 0 then (* which we read some data *)
            writeln ('can't open the file!') ;

            for rn:=1 to 6 do      (* output file f contents *)
                [seek (f,rn) ;      (* to terminal screen *)
                get (f) ;
                for j:=1 to size do
                    write (dl[j]) , writeLn] ;
                close (f) ] ;
            close (fml)
        end;
end;

```

文件namlenth.dat的内容:

```

a.dat    10
b.dat    20
c.dat    30
d.dat    80

```

四个文件的输出结果:

```

a.dat:
1111111111
2222222222
3333333333
4444444444
55555
6666666666
b.dat:
aaaaaaaaaaaaaaaaaaaaa
bbbbbbbbb
ccccccccccccccccccc

```

```

dddddddddddddddddddd
eeeeeeee
fffffffffffffffffff
c.dat:
11111111111111111111111111111111
//////zzzzz///zzzzzzzzzzzz///
0000000000000000
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
'>>>>>>>>>>>>>>>>>>>>>>>>>
yyyyyyyyyyyyyyyyyyyyyy
d.dat:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
EEEEEEEEEE
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

```

在这个程序中，说明了一个“假”的文件f，其目的在于得到一个文件控制块FCB，但它不能代表任何一个实际文件，因为它被定义为记录属于STRING（1）的文件，而四个实际文件的记录长度都不为1，要让它代表某一个实际文件，就要“修正”记录长度值，还要给它一个“实际”的文件变量缓冲区f^。这些正是OPEN过程要实现的主要功能。具体解释到稍后部分给出，请大家首先把自己的注意力集中到该程序的基本功能，或者说该程序的“外”特性上。

这个程序要读的四个不同长度的STRING类型的文件的名字和记录长度登录在namlenth.dat文件中。对四个文件的操作用for语句控制。具体步骤是，用OPEN过程读来一个文件的文件名和记录长度，并把该长度值记到FCB的相应单元中，并指定变量d1为f^的实际内存区。然后依次读来该文件的前六个记录的内容，并将其显示在终端屏幕上，再关闭文件。

这个程序能说明些什么问题呢？或者说，我们想用这个例子为读者提供些什么信息呢？回答是：

- 第一，对这个程序用到的四个记录长度不等的文件，没有像通常那样说明四个文件变量，而是借助于唯一的一个文件变量f进行操作的。当文件数目为4时，这个优点不明显，设想文件数目为几十个，上百个（在数据库管理系统中，这是必然现象），这个用法就成为摆脱困境的必要手段。
- 第二，每个文件的记录长度和文件变量缓冲区的实际地址，是可以在打开文件之前指定的，并写入到文件控制块FCB(File control Block)中。在打开文件后，读写文件都受

这两个值的约束。对已打开的文件，在其打开期间不能执行上述操作。

第三，每个文件的控制块都是相同的类型，作为某些文件操作的过程和函数，如 RESET、EOLN 等，它们的参数说明均可写成

(VAR F:FCBFQQ)

的形式。这里的FCBFQQ就是文件控制块的类型，是PASCAL语言用的系统关键字。在PASCAL语言中，不同结构的文件，一般应有各自的FCB。我们在上面的程序中，实际上应用了一个变通的办法，让四个不同结构的文件，依次使用了同一个FCB。

第四，上述程序中的四个文件的区别，仅表现为记录长度不等，但每个记录都属于一定长度的STRING类型。这样做的目的，只是为了可以用统一的语句在终端屏幕上显示记录内容，程序最简洁。在实际应用中，每个文件记录的组成情况，如域的数目，每个域的类型和长度都可以按需要定义。这对打开文件和读出一个记录的办法均无影响，差别仅表现在对读出的一个记录具体分解成不同的域，对每一个域的处理和使用可以不同。由此得出结论，上述程序中应用的技术是通用的。

第五，在用常规办法使用文件时，并不要求用户指定文件记录长度和文件变量缓冲区地址，那是由PASCAL编译程序处理用户程序中的每个文件变量说明部分时自行解决的。当我们采用上面程序中的办法处理文件时，每个文件不再单独说明，编译程序也就不能找出这两个值，而必须由用户给出，其值的正确性也就由用户自行负责，给的不正确，将不会产生正确的运行结果。

为了加深对上述五点论述的认识，我们将把上面的程序稍加改动，变成先写入一个文件的几个记录，然后再读出来检查的程序。对同一个文件，写入和读出试用不同的记录长度和文件变量缓冲区，再检查运行结果。下面是该程序的清单。

```
program filusel (input,output) ;
type a=string (80) ;
var
    f : file of a;      (* File's record length is 80 *)
    d1, d2:a;           (* Two data buffers *)
    fads:ads of fcbfqq; (* FCB address *)
    brr : ads of adrmem; (* Char array address *)
    size, rn, w:word;
    wr:ads of word;
    i :integer;
    datar:adr of a;      (* Using for giving d1, d2 address *)
procedure open;
begin
    write('file name? '); readln (input, f) ;
    f.mode:=direct; f.trap:=true;
    write ('size? ');
    readln (size) ;
    fads:=ads f;
```

```

    wr.s:=fads.s;           ! Write file record size into
    wr.r:=fads.r+10;        ! FCB's proper location
    wr^:=size;
    brr.s:=fads.s;
    brr.r:=fads.r+24;       ! Write file buffer address into
    datar:=adr dl;         ! FCB's proper location
    brr^.r:=datar.r;
end;

begin (*main program*)
    (*open the file f into which
       we write some data*)
    open;
    rewrite (f) ;
    (*input some data into file f*)
    repeat
        write('record number ? '); readln (rn) ;
        if rn< >0 then
            [seek (f,rn) ;
             write('record content? ');
             readln (dl);          (*f^:=dl*)
             put (f)]
        until rn=0;      close(f);

        (*open the file f from which
           we read some data *)
        open;
        reset(f);

        (*output file f contents to terminal screen*)
        repeat
            write ('rn ? '); readln (rn) ;
            if rn< >0 then
                [seek(f,rn) ; get (f) ;
                 for w:=1 to size do write (dl[w]) ;
                 writeln
                ]
            until rn=0;
            close (f)
        until rn=0;
    end.

```

在这个程序中，OPEN过程是从终端上读入文件名和文件记录长度，运行时，用户可以按自己的意图给出相应的回答。我们又指定文件 f 为 DIRECT 访问方式，用 REW-

RITE打开，可以指定写入文件中的记录内容和写入的位置（由记录号决定）。读出时，可按指定的记录号读出相应的记录。该程序运行过一次之后，可以只用RESET打开文件，并指定与原来记录长度不同的SIZE值读出某些记录，看一看运行的结果。此外，还可以指定d2作为文件变量缓冲区，或既不用d1，也不用d2，而直接使用F^，看一看运行的结果，体会文件的记录长度、变量缓冲区对读写文件的约束作用。

对上面讲过的内容和给出的程序例子，还可以进行规范，例如，在一个UNIT中实现三个过程OPEN、READREC和WRITEREC，分别用于打开任意的文件，读其记录到指定的内存地址，或把内存中的内容写到文件中。

正像在接口文件中说明的那样，对OPEN过程要给文件变量名和文件记录长度，并指定文件为DIRECT读写方式，用REWRITE过程打开，此时，若指定的文件已经存在，原有内容不会丢失，指定的文件不存在，会新建一个。对读写过程READREC和WRITEREC除了要给文件名外，还必须指定读或写的记录号和内存缓冲区变量。这个变量的地址，是在过程内被写到文件控制块FCB的相应单元的，在需要时，完全可以把文件的不同的记录的内容，不经特定的文件变量缓冲区F^而直接读到内存中几个指定的区域。

下面给出这个接口文件的清单：

```
INTERFACE;
```

```
UNIT FILE1 (OPEN, READREC, WRITEREC);
```

```
  PROCEDURE OPEN(VAR F:FCBFQQ; SIZE:WORD);
```

```
    (* This procedure opens a direct data file with a record
       length of SIZE.
```

```
       Before calling it, user must assign the file parameter f
       to a file name by useing ASSIGN procedure or READFN procedure.
       User can identify a general file variable such as
       "F:FILE OF STRING (10);" to access a data file with any
       record length. *)
```

```
  PROCEDURE READREC(VAR F:FCBFQQ; RN:WORD; VAR BUF:
                     STRING)
```

```
    (* F : File variable identified in user's program.
       RN :Record number inquired.
       BUF:Buffer variable identified by user. *)
```

```
  PROCEDURE WRITEREC(VAR F:FCBFQQ; RN:WORD; VAR BUF:
                      STRING);
```

```
    (* The same as that in READREC *)
```

```
BEGIN
```

```
END;
```

下面是实现三个过程的UNIT的实现部分：

```
  (* $INCLUDE:'FILE1.INF' *)
```

```
IMPLEMENTATION OF FILE1;
```

```
  VAR FADS:ADS OF FCBFQQ;
```



```

WR:ADS OF WORD,
BRR:ADS OF ADRMEM,
CR:ADR OF CHAR,

```

```

PROCEDURE OPEN,
BEGIN
    F.MODE:=DIRECT    ,
    FADS  :=ADS F      ,
    WR.S  :=FADS.S     ,
    WR.R  :=FADS.R+10,
    WR^   :=SIZE       ,
    REWRITE (F)

```

```

END,

```

```

PROCEDURE READREC,
BEGIN

```

```

    IF RN > 0 THEN
        [FADS :=ADS F,
         WR.S  :=FADS.S,      WR.R :=FADS.R+20,
         BRR.S :=FADS.S,      BRR.R:=FADS.R+24,
         CR    :=ADR BUF[1],  BRR^.R:=CR.R,
         SEEK (F, RN) ,      GET (F) ]

```

```

END,

```

```

PROCEDURE WRITEREC,
BEGIN

```

```

    IF RN > 0 THEN
        [FADS :=ADS F ,
         WR.S  :=FADS.S,      WR.R :=FADS.R+20,
         BRR.S:=FADS.S,      BRR.R:=FADS.R+24,
         CR    :=ADR BUF[1],  BRR^.R:=CR.R,
         SEEK (F, RN) ,      PUT (F) ]

```

```

END,

```

```

BEGIN
END.

```

下面给出应用上述UNIT的一个程序例子。程序的功能，是把用file of integer类型定义的一个文件的内容，快速读到用类型array [1..6] of string (512) 定义的数组中。再通过地址类型把这数组的3072个字节中的内容，作为1536个整型量输出到终端屏幕上，同时把数组内每个整数值加上512后再写回文件。为了看得更明白，我们先用常规办法建立与写文件，只在读文件的部分使用 UNIT 过程提供的三个过程。文件的内容是用 for i:=1 to 1536的办法把 i 值写入的，检查输出结果是否正确就变得简单了。读者也可以

把这后一部分用常规办法实现，或用另外的记录长度处理，有兴趣者可以试一试。

为了检查重写后文件的内容，我们给出了另一个标题为TEST的程序。两个程序的源清单如下。

```
(* $include:'file1.inf' *)
program filuse3 (input, output) ,
  uses file1 (open, readrec, writerec) ,

const
  recsize=512,
var
  f:file of integer,
  ar:array[1..6] of string (recsize) ,
  ad:adr mem,
  adi:adr of integer,
  i:integer,

begin
  assign (f,'test.dat') , rewrite (f) ,
  for i:=1 to 1536 do
    [f^:=i, put (f) ],
  close (f) ,
  ad:=adr ar,    adi:=ad,
  open (f, recsize) ,
  for i:=1 to 6 do readrec(f, wrd(i), ar[i]) ,

  for i:=1 to 1536 do
    [write (adi^:6) , adi^:=adi^+recsize,
     write (adi^:6) ,
     adi.r :=adi.r+ 2,
     if i mod 6 = 0 then writeln],

  for i:=1 to 6 do writerec (f, wrd(i), ar[i]),
  close(f),
end.

program test (input, output) ,
var
  f:file of integer,
  i:integer,
begin
  assign (f, 'test.dat'), reset (f) ,
```

```

    for i:=1 to 1536 do
      [write(f^:6), get(f),
        if i mod 12= 0 then write^n^,
      close(f)
    end.

```

下面给出DOS系统对设备和文件的有关说明，直接打印出有关内容，不进行翻译和说明。

{MS-System File Control Block, DOS Specific Version}

INTERFACE, UNIT

```

    FILKQQ (FCBFQQ, FILEMODES, SEQUENTIAL, TERMINAL,
            DIRECT,
            DEVICETYPE, CONSOLE, LDEVICE, DISK,
            DOSEXT, DOSFCB, FNLUQQ, SCTRLNTH) ,

```

CONST

```

    { $INCONST:MS-DOS 1 for MS-DOS, 0 for CP/M 80 or 86}
    FNLUQQ=21,           {length of a DOS filename}
    { $IF MS-DOS $ THEN}
    SCTRLNTH=512,        {length of a disk sector}
    { $ELSE}
    SCTRLNTH=128,        {length of a disk sector}
    { $END}

```

TYPE

```

    DOSEXT=RECORD          {DOS file control block extension}
      PS [0]:BYTE,         {boundary byte, not in extension}
      FG [1]:BYTE,         {flag; must be 255 in extension}
      XZ [2]:ARRAY [0..4] OF BYTE, {pad, internal use}
      AB [7]:BYTE,         {internal use for attribute bits}
    END,

```

```

    DOSFCB=RECORD          {DOS file control block (normal) }
      DR [00]:BYTE,        {drive numb, 0=default, 1=A etc}
      FN [01]:STRING(8),   {file name-eight characters}
      FT [09]:STRING(3),   {file extn -three characters}
      EX [12]:BYTE,        {current extent, lo order byte}
      E2 [13]:BYTE,        {current extent, hi order byte}
      S2 [14]:BYTE,        {size of sector, lo order byte}

```

RC [15]:BYTE,	{size of sector, hi order byte}
Z1 [16]:WORD,	{file size, lo word, readonly}
Z2 [18]:WORD,	{file size, hi word, readonly}
DA [20]:WORD,	{date, bits: DDDDDMMMMYYY\ \ \ \}
DN [22]:ARRAY [0..9] OF BYTE,	{reserved for DOS}
CR [32]:BYTE,	{current sector (within extent)}
RN [33]:WORD,	{direct sector number (lo word)}
R2 [35]:BYTE,	{direct sector number (hi byte)}
R3 [36]:BYTE,	{DSN hi byte iff sect size<64}
PD [37]:BYTE,	{pad to word boundary, not DOS}

END,

DEVICETYPE=(CONSOLE, LDEVICE, DISK) : {physical device}
 FILEMODES=(SEQUENTIAL, TERMINAL, DIRECT) : {accessmode}
 TYPE

FCBFQQ=RECORD {byte offsets start every field comment}
 {fields accessible by Pascal user as <file variable>.<field>}

TRAP:BOOLEAN,	{00 Pascal user trapping errors if true}
ERRS:WRD(0)..15,	{01 error status, set only by all units}
MODE:FILEMODES,	{02 user file mode, not used in unit U}
MISC:BYTE,	{03 pad to word bound, special user use}
{fields shared by units F, V, U, ERRRC/ESTS are write-only}	
ERRC:WORD,	{04 error code, error exists if nonzero}
	{1000..1099: set for unit U errors}
	{1100..1199: set for unit F errors}
	{1200..1299: set for unit V errors}
ESTS:WORD,	{06 error specific data usually from OS}
CMOD:FILEMODES,	{08 system file mode, copied from MODE}
{fields set /used by units F and V, and read-only in unit U}	
TXTF:BOOLEAN,	{09 true: formatted/ASCII/TEXT file}
	{false: not formatted/binary file}
SIZE: WORD,	{10 record size set when file is opened}
	{DIRECT:always fixed record length}
	{others:max buffer variable length}
MISB: WORD,	{12 unused, exists for historic reasons}
OLDF: BOOLEAN,	{14 true: must exist before open, RESET}
	{false:can create on open, REWRITE}
INPT: BOOLEAN,	{15 true:user is now reading from file}

	{false:user is now writing to file}
RECL:WORD,	{16 DIRECT record number, lo order word}
RECH:WORD,	{18 DIRECT record number,hi order word}
USED:WORD,	{20 number bytes used in current record}

{field used internally by units F and V not needed by unit U}

LINK:ADR OF FCBFQQ; {22 DS offset address of next open file}

{fields used internally by unit F not needed by units V or U}

BADR:ADRMEM,	{24 ADR of buffer variable (end of FCB)}
IMPF:BOOLEAN,	{26 true if temp file; delete on CLOSE}
FULL:BOOLEAN,	{27 buffer lazy evaluation status, TEXT}
MISA:BYTE,	{28 unused, exists for historic reasons}
OPEN:BOOLEAN,	{29 file opened; RESET/REWRITE called}

{field used internally by unit V not needed by units F or U}

UNIT:INTEGER,	{30 Unit V's unit number always above 0}
LNDF:BOOLEAN,	{32 last operation was the ENDFILE stmt}

{fields set/used by unit U, and read-only in units F and V}

REDY:BOOLEAN,	{33 buffer ready if true, set by F/U}
BCNT:WORD,	{34 number of data bytes actually moved}
EORF:BOOLEAN,	{36 true if end of record read, written}
EOFF:BOOLEAN,	{37 end of file flag set after EOF read}

{unit U (operating system) information starts here}

NAME:LSTRING (FNLUQQ),	{38 DCS filename(D:NNNNNNNNN.XXX)}
DEVT:DEVICETYPE,	{60 device type, accessed by file}
RDFC:BYTE,	{61 function code, for device GET}
WRFC:BYTE,	{62 function code, for device PUT}
CHNG:BOOLEAN,	{63 true if sbuf data was changed}
SPTR:WORD,	{64 index to current byte in sbuf}
LNSB:WORD,	{66 number of valid bytes in sbuf}
DOSX:DOSEXT,	{68 extend DOS file control block}
DOSF:DOSFCB,	{76 normal DOS file control block}
NEOF:BOOLEAN	{114 true if eoff is true next get}
FNER:BOOLEAN,	{115 true if pfnuqq filename error}
SBFL:BYTE,	{116 max textfile line len in sbuf}
SBFC:BYTE,	{117 number of chars, read to sbuf}
SBUF:ARRAY [WRD(0)..SCTRLNTH-1] OF BYTE,	{118 sect buffer}
PMET:ARRAY [0..3] OF BYTE,	{118+ctrlnth reserved pad}

```

    BUFF:CHAR,                                {122+strlen(buffer var)}
    {end of section for unit U specific OS information}
END;
END;

```

10.2 磁盘的盘区分配及读写

一、磁盘扇区的使用规则

磁盘的盘区分配及物理读写过程都以扇区为单位。

DOS系统管理的磁盘，每个扇区由512个字节组成。对WANG IPC机所配的30MB容量的硬盘，其规定是：

第一个扇区中（逻辑扇区号为0）放的是系统的引导程序。

第二到第九个扇区（扇区编号为1~8）放的是文件分配表FAT的第一个拷贝。

第十到第十七个扇区（扇区编号为9~16）放的是文件分配表的第二个拷贝。

第十八到第二十五个扇区（扇区编号为17~24）用作为磁盘的根目录区。

余下的扇区（编号从25开始）用来存放系统和用户的文件，是数据区（包括目录区）。数据区分配的单位叫作簇，每个簇由32个扇区（即16KB存储空间）组成。

二、文件分配表(FAT)的使用格式

整个磁盘的使用情况登记在文件分配表中，它占用连续的八个扇区，即由4096个字节组成。使用时，用每1.5个字节（即12bits）表示一个簇号。这样算下来，它可以给出4096/1.5个（即2730个）不同的簇号。对30MB硬盘，系统中真正要用的是2042个（30MB/16KB）簇号。我们常称这每1.5个字节为FAT中的一个登记项。

FAT中的第一和第二个登记项（共三个字节）保留给系统用。有用的登记项的编号从002开始。登记项的内容为FF0到FF7时，表明是保留簇，其中FF7表示簇区已坏，不能使用。FF8到FFF用来表示文件结束的簇。一个文件占用多个簇时，这些簇是以链表形式联系在一起。

每个簇号对应着磁盘上一个16KB的簇空间，该空间的起始扇区编号的计算公式为：

$$\text{扇区编号} = 25 + (\text{簇号} - 2) * 32$$

这里的25是1个引导扇区，16个FAT扇区，8个根目录扇区之和。簇号0和1系统保留专用，可用簇号从2开始，每个簇由32个扇区组成。

三、文件目录的组成格式

每一个文件的目录由32个字节组成。根目录分配在17至24号八个扇区内，最多可登入的文件目录数为：

$$512 \times 8 \div 32 = 128$$

每个文件的目录具体组成格式如下：

字节编号	登录的内容	说	明
0 ~ 7	文件的名字	这个域的第一个字节还被用来表示文件状态。	
		00H	结束目录查找标志
		E5H	文件删除标志
		2EH	用于一个记录, 当第二个字节仍为2EH时, 簇区域为其上级目录的簇号
8 ~ 10	文件扩展名		
11	文件属性	01H	只读文件
		02H	隐藏文件
		04H	系统文件
		08H	目录的前11个字节中包括卷标, 只用在根目录段
		10H	为子目录
		20H	档案位, 建备份有用
12 ~ 21	保留	格式如下:	
22 ~ 23	文件建立或修改的时间	15 ~ 11	10 ~ 5
		hh	mm
		0 ~ 23	0 ~ 59
		小时	分
			两秒
			单位
24 ~ 25	日期	15 ~ 9	8 ~ 5
		年	月
			日
		0 ~ 119	1 ~ 12
			1 ~ 31
		代表1980 ~ 2099	
26 ~ 27	开始的簇号		
28 ~ 31	以字节数表示的文件长度		
	第一个字是低位部分		

四、磁 盘 读 写

DOS系统支持按磁盘的绝对扇区号进行磁盘读写, 是通过INT 25H和INT 26H实现的。调用前, DX寄存器中给出起始扇区号, BX中给出内存缓冲区地址, CX中给出读入的扇区数, AX寄存器的低位字节AL中给出磁盘驱动器编号(A盘编号为0, B盘为1, ..., E盘为4)。我们可以用这两种中断调用实现PASCAL程序中读写磁盘扇区的过程。

在进行了上述说明之后, 我们就能在自己的PASCAL程序中直接对硬磁盘进行管理与操作了。但这里要提请读者注意, 在您尚未非常准确地掌握本节介绍的内容之前, 或者在您的应用程序中没有必要直接对磁盘直接操作的情况下, 请不要直接操作磁盘。因为在这种用法中, 若程序中出现错误, 会导致丢失磁盘上有用信息, 甚至破坏整个磁盘的正常工作过程, 使机器再不能运行。出现这种情况后, 若不能正确恢复, 就只能重新安装系统, 硬盘上的信息全部丢失。

下边给出的第一个程序例子, 是首先读出硬盘文件分配表FAT的内容到数组SEC, 计算出其中每个有效登记项所对应的簇号, 并把计算结果写进数组FAT中。这是通过调用过程FATDECODE完成的。然后修改文件分配表中部分内容(请记下修改中的全部情况

以备恢复), 再把修改后的FAT的全部内容写回原来位置, 这是通过调用FATENCODE过程实现的。

程序中用到了按块读写磁盘的两个过程DSKI和DSKO, 它们是用汇编语言实现的外部过程, 实现的办法就是在本节的内容4中说过的。它们的参数a, 是一片内存区的起始地址。该片内存区的大小, 正好能放下 $512 \times 8 = 4096$ 个字节, 即放得下FAT的全部内容。参数st和num是被读或被写的磁盘区的起始扇区号和读入的总的扇区数。在我们的程序中, 可以选ST为1和NUM为8, 就能读来FAT的头一个拷贝的内容。

在fatdecode过程中, 求得每个登记项的内容, 是分三步实现的:

第一步, 用簇号求出该登记项在数组SEC中的高位地址,

第二步, 把找到的位置及后面一个位置两个字节的内容合成一个字,

第三步, 取合成一个字中对应的12个二进制位所给出的值, 并将其写入数组FA的相应元素中。

如果用户要求显示数组FA中的内容, 还可以将其显示到终端屏幕上。显示过程中采用16进制形式输出, 该程序内部实现了一个按16进制形式输出一个字类型数据的过程。

在过程fatencode中, 有实现把输入的一个整数变换成12位二进制形式的数、并将其写到文件分配表相应位置的功能。最后写文件分配表, 给出了同时写到两个拷贝上去的语句, 两份文件分配表在正常情况下的内容应该保持一致。下面是这个程序的清单。程序之后给出的是汇编语言实现的外部过程的程序清单。

检查显示在终端屏幕上的FAT的内容, 特别是与有关的文件目录一起看, 会加深对FAT功能的认识。

```
( * $warn - * )
program fat (input, output);
type
    fat-sector=array[0..4095] of byte;
    fat-table=array[0..2730] of word;
var
    sec:fat-sector;
    fat:fat-table;

    procedure DSKI( var a:fat-sector; st,num:integer); extern;
    procedure DSKO(var a:fat-sector; st,num:integer); extern;

    procedure hexout(i:word);
        var j:integer;
        procedure aout(i:word);
        begin
            if(i>9) and (i<16) then [i:=i mod 10;
                                    i:=i-65;
                                    write(chr(i))]
            else write(i:1
```



```

    end,

begin
    write(' ');
    j:=i div 256; i:=i mod 256;
    aout(j);
    j:=i div 16; i:=i mod 16;
    aout(j); aout(i);
    write('H')
end,

procedure fatdecode(var fa:fat-table;
                    var sector:fat-sector);

var
    i, j, limit, offset:integer;
    work:word;      ch:char;
    f:text;

begin
    writeln('This routine is going to display',
           'and rewrite FAT of Winchester');
    writeln('Before that, You must know where',
           'is Fat in, and how long? ');
    writeln('Now please input it---');
    writeln,

    write('THE START SECTOR---');      readln(i);
    WRITE('TOTAL OF SECTOR---');      readln(j);
    DSKI (sector, i, j);
    writeln('disk read done!');
    for i:= 0 to 2042 do
        [offset:= (i*3) div 2;
         work:=byword(sector[offset+1], sector[offset]);
         if odd (i) then work:=work div 16
           else work:=work mod 4096;
         fa[i]:=work
        ],

    write('Are you going to Display the FAT? ',
           'It is some what long (Y/N) ');
    readln (ch);
    if (ch='y') or (ch='Y') then
        [writeln('--File Allocation Table of w-Disk--');
         for i:= 0 to 2042 do

```

```

        [write ('*',i:3,' (') ;
          hexout (i) ; write (') ') ;
          hexout (fa[i]) ;
          if i mod 5 = 0 then writeln,
        ]
      ]
end;

procedure fatencode (var fa:fat-table;
                     var sector:fat-sector) ;

var
  i, j :integer;
  limit:integer;
  offset:integer;
  work:word;
begin
  writeln ('Now you can change the data of FAT,',
           'END by prass 0') ;
  write ('Which NO. fat will be change? —'),
  readln (i) ;
  while i >= 1 do
    [WRITE ('The FAT old value is—', fa[i]:5, ' ') ;
     hexout (fa[i]) ;
     writeln,
     write ('New Value ? ') ;
     readln (j) ;
     fa[i] := j;
     write ('Which NO. fat will be change?—') ;
     readln (i) ;
    ],
  for i := 0 to 2042 do
    [ offset := (i * 3) div 2 ;
      work := byword (sector[offset+1], sector[offset]) ;
      if odd (i)
        then work := work mod 16 + fa[i] * 16
        else work := (work div 4096) * 4096 + fa[i];
      sector [offset] := lobyte (work) ;
      sector [offset+1] := hibyte (work) ;
    ],
  dsko (sector, 1, 8) ;
  {dsko (sector, 9, 8) ;}

```

```

    end,

begin  (*Main program*)
    fatdecode (fat, sec) ;
    fatencode (fat, sec)
end.
DSEG SEGMENT PUBLIC 'DATA'
DSEG ENDS
ASSUME CS:CSEG, DS:DSEG, ES:DSEG, SS:DSEG
CSEG SEGMENT 'CODE'
PUBLIC dski
dski    PROC        FAR
        PUSH        BP
        MOV         BP,        SP
        MOV         DX,        [BP+ 8 ]
        MOV         BX,        [BP+0A1H]
        MOV         AL,        4
        MOV         CX,        [BP+ 6 ]
        INT         25H
        POPF
        POP         BP
        RET         6
dski    ENDP
PUBLIC dsko
dsko    PROC        FAR
        PUSH        BP
        MOV         BP,        SP
        MOV         DX,        [BP+ 8 ]
        MOV         BX,        [BP+0A1H]
        MOV         AL,        4
        MOV         CX,        [BP+ 6 ]
        INT         26H
        POPF
        POP         BP
        RET         6
dsko    ENDP
CSEG ENDS
END

```

10.3 文件目录的直接操作

从上一节我们知道，DOS系统支持对磁盘的按块读写功能。如果我们知道文件目录区的绝对扇区号，就可以比较容易地实现对文件目录的直接操作。下面，我们以使用软盘为例，介绍在PASCAL程序中如何对磁盘的根目录进行查找、读写和修改。

DOS系统中，软盘的扇区通常是按如下规则分配的：

第0扇区为系统引导块。

第1扇区和第2扇区为文件分配表FAT。

第3扇区和第4扇区为FAT的备份。

第5扇区到第11扇区为磁盘的根目录区。

其余扇区为系统和用户的文件区。

根据上述磁盘分区情况，我们要对软盘根目录区进行操作，只需从它的第5扇区开始读入7个扇区，就得到了根目录区中的全部内容，并可以对其进行修改等操作。

我们来看一个具体的程序例子。其功能是用来恢复由于误操作而刚被删掉的文件。只要在文件被删后尚未进行其它有关文件方面的操作时，被删掉的文件是可以恢复的。

```
( * $warn- * )
program dirop (input, output);
type
    data-type = array [0..4095] of byte;
var
    start, total: integer;
    dl          : data-type;
    dir, i,      : integer;
    equal        : boolean;
    name         : array [0..7] of char;
procedure dski (var ds: data-type; a, b: integer); extern;
procedure dsko (var ds: data-type; a, b: integer); extern;
begin
    start := 5; (* the first sector of root *)
    total := 7; (* the number of root *)
    dski (dl, start, total); (* read in *)
    write ('PLEASE INPUT THE FILE NAME YOU JUST DELETED: ');
    for i := 0 to 7 do
        read (name [i]);
    readln;
    dir := 0; equal := false;
    while (dl [dir] > 0) and (not equal) do
        if (dl [dir] = 20)
        then
```

```

        [equal:=true; j:=1;
        for i:= (dir+1) to (dir+7) do
            [if dl[i] < >ord (name[j])
                then equal:=false;
                j:=j+1;
            ]
        ],
        dir:=dir+32
    ],
    if equal then
        [dir:=dir-32; dl[dir]:=ord (name[0]);
        dsko (dl, start, total); write ('success!');
        ]
    else write ('FILE NAME ERROR!')
end.

```

这个程序用到的外部过程，与前一节中用到的基本相同，只是AL中使用的设备号变为0（代表A盘），程序清单给出如下：

```

DSEG SEGMENT PUBLIC 'DATA'
DSEG ENDS
ASSUME CS:CSEG, DS:DSEG, ES:DSEG, SS:DSEG
CSEG SEGMENT 'CODE'
PUBLIC dski
dski    PROC    FAR
        PUSH    BP
        MOV     BP,    SP
        MOV     DX,    [BP+8]
        MOV     BX,    [BP+0AH]
        MOV     AL,    0
        MOV     CX,    [BP+6]
        INT     25H
        POPF
        POP     BP
        RET     6
dski    ENDP
PUBLIC  dsko
dsko    PROC FAR
        PUSH BP
        MOV BP, SP
        MOV DX, [BP+8]
        MOV BX, [BP+0AH]

```

```

MOV AL, 0
MOV CX, [BP+6]
INT 26H
POPF
POP BP
RET 6

```

```

dsko ENDP
CSEG ENDS
END

```

程序首先将根目录区读入内存缓冲区d1。由于删除文件操作仅将相应目录项的第一字节中的值改16#E5（十进制为229），恢复文件只需将字改回来就行了。如程序清单中所表明的。

实际上，读出目录区以后，我们也可以对目录区的各项进行修改，其中最常用的大概就是修改文件的属性。例如，要使文件成为隐含的，只需将目录项中的属性改为02即可。下面是改文件为隐含文件的程序清单：

```

(* $warn - *)
program dirop1 (input, output);
type
  data-type=array [0..1095] of byte;
var
  start, total:integer;
  d1          :data-type;
  dir, i, j   :integer;
  equal       :boolean;
  name        :array [0..7] of char;
  procedure dski (var ds:data-type; a,b:integer); extern;
  procedure dsko(var ds:data-type; a,b:integer); extern;
begin
  start:=5; (*the first sector of root*)
  total:=7; (*the number of root*)
  dski (d1, start, total); (*read in *)
  write ('PLEASE INPUT THE FILE NAME YOU WANT TO HIDE-
        DEN:');
  for i:=0 to 7 do
    read (name [i]);
  readln;
  dir:=0; equal:=false;
  while (d1 [dir] >0) and (not equal) do
    [equal:=true; j:=0;
      for i:=dir to (dir+7) do
        [if d1 [i] < >ord (name [j])

```

```

        then equal:=false;
        j:=j+1;
    ],
    dir:=dir+32
],
if equal then
    [dir:=dir-32; d1[dir+11]:=2;
    dsko(d1, start, total); write('success!');
    ]
    else write('FILE NAME ERROR!')
end.

```

修改指定的文件目录的其它各项内容的办法与此相同，例如，完成改名或删除文件，改变文件建立或修改的日期等等。查找文件目录的办法在这个程序中已用到了。

以上程序都是通过软中断INT25H和INT26H实现的。除此之外，DOS还提供了其它一些对文件或目录进行操作的系统调用，包括：

编 号	功 能
4EH	查找第一个文件；
4FH	查找下一个文件；
43H	置/取文件属性；
57H	置取日期/和时间；
56H	文件更名；
39H	建立一个子目录；
3AH	删除一个子目录；
3BH	改变当前目录；
47H	取当前目录路径名。

下面，我们给出一个用上述系统调用来实现隐含文件的程序，读者可以将它与前一个程序进行比较，体会它们的差异之处。

```

program filemode (input, output),
type
    str=string (13),
var
    filename, ss:str;
    i, j:integer;
    procedure md (var ss:str; k:integer); extern;
begin
    write ('please input file name:'); readln(filename);
    writeln('--- please input file mode---');
    writeln ('0 bit read only, 1 bit hidden file');
    writeln ('2 bit system file, 3 bit volume ID');
    writeln ('4 bit directory, 5 bit file created or writer to');

```

```

write ('———'),
readln (i),
ss [13] := chr (0),
for j:=1 to 15 do ss [j] := filename [i],
md (ss, 7);
md (ss, i)
end.

```

```

dseg segment 'data'
dataarea db 200 dup (')
dseg ends

assume cs:cseg, ds:dseg, ss:dseg
cseg segment 'code'
public MD
MD      proc far
        push bp
        push ds
        mov bp, sp
        mov dx, [bp+0ah]
        mov cx, [bp+08h]
        mov ax, 4301H
        int 21h
        pop ds
        pop bp
        ret 4
MD      endp
cseg    ends
end

```

以上，我们用软中断和功能调用分别实现了对软盘根目录的直接操作。至于对硬盘目录和子目录的操作，与此类似，留给读者自己解决。

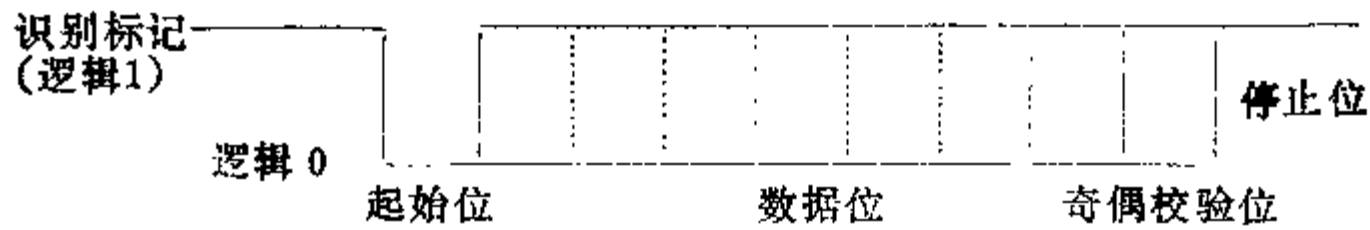
把本节内容和前一节内容结合起来，就能在PASCAL中自行分配磁盘空间，建立子目录区或文件（子目录区是一种特殊的文件），在文件目录上进行必要的操作，也可以从根目录开始，沿给出的路径名找到相应目录或文件，并且可以对该文件实现按扇区读写等一系列操作。我们在此不再给出例子和进一步说明，留给读者去自行解决。

10.4 利用串行口实现计算机之间的通信

可以利用微机系统提供的串行口，直接实现两台近距离微机之间的通信。在两台微机距离较远时，要加进调制解调器才能实现串行口通信。采用串行通信有很多优点。一是串行接口标准化程度高，对不同系列或型号的计算机几乎可不做任何匹配就能直接接通。二是串行通信可以节省庞大的电缆开支，在许多情况下，就直接选用已有的电话线，这是简

便、经济、快速地建立两台计算机之间通信的良好途径。

串行通信，是指被发送或接收的数据和各种控制信息，都在单根数据传送线上以二进制信息形式逐位串行传送。这些二进制位信息的关系如下图所示：



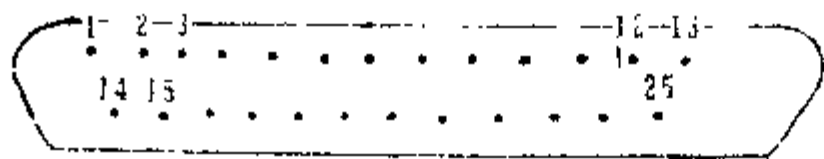
以传送字符为例，数据位通常由 7 个二进制位组成，它的内容是每个字符的 ASCII 码。奇偶校验用 1 位，当选用奇校验时，使 7 位数据位和校验位中的 1 的个数凑成奇数；当选用偶校验时，使这八位中 1 的个数凑成偶数；当用一个字节存放一个字符时，奇偶校验位正好用一个字节的最高一个二进制位表示。在被传送的数据（包括校验位）前后都加进了控制信息。数据之前使用了一个起始位，作为发送方通知接收方准备接收数据的同步信号。在数据之后，要用 1、1.5 或 2 位的停止位，它给出的是发送完一个字符数据后，可以开始下一次传送的最短间隔时间，在这一间隔时间内，传送线一直处于逻辑 1 状态。在开始下一次传送时，线的状态首先要从 1 变为 0 状态，这就是起始位的功能。接收方要靠接收到起始位才开始接收一个字符，否则将一直处于等待状态，这就是异步通信的含义。

数据信息位和控制信息位的宽度，取决于数据传送的波特率，即传送线上每秒传送多少个二进制位。接收与发送双方必须设置相同的波特率，常用的有 300、600、1200、2400、4800 和 9600 等等。奇偶校验位是选用的，可选奇校验、偶校验，或根本不用校验。在这后一种情况下，停止位将直接跟在数据位之后。此外，数据位的位数、停止位的个数也可以选不同的值，但发送方与接收方必须设置相同的值。

直接使用 RS232 串行口进行通信是困难的。为此，PC 机中都使用了一个通用异步接收发送器（UART）的专用微处理器，即 8250 组件。这样，短距离的异步串行接口通信的连接关系是：

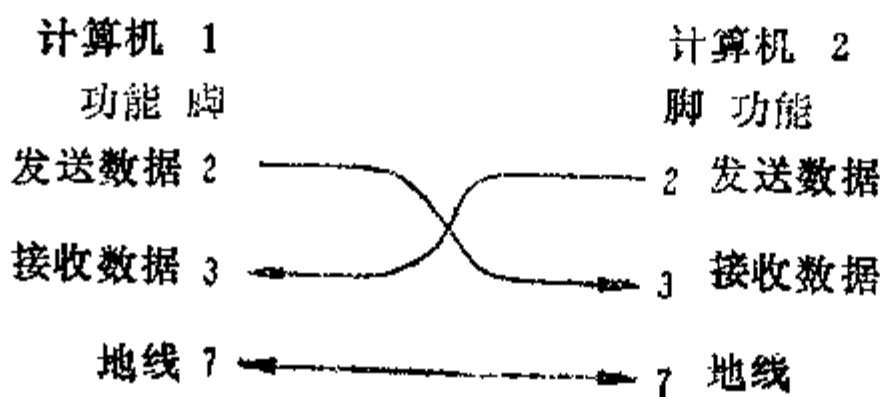


RS232 接口的输出电平是：逻辑 0 用 +3 ~ +15V 表示，逻辑 1 用 -3 ~ -15V 表示。输出用的是标准 25 芯 D 型插座。



插座脚 2 用于发送，脚 3 用于接收，脚 7 是信号地。

在做两台计算机间的连线时，要把计算机 1 的插头脚 2 与计算机 2 的插头脚 3 相连，计算机 2 的插头脚 2 与计算机 1 的插头脚 3 相连，不能用 2 ↔ 2、3 ↔ 3 的方式连接。即：



PC机的OUT和IN指令可以用于对UART的操作。在BIOS中还提供了使用UART的功能调用，这些内容这里不介绍，有兴趣的读者请参阅PC机的有关技术资料。我们给出的程序中，是用OUT与IN指令直接对串行口进行控制的。

下面给出实现串行口通信的一个简单程序。这个程序是原理性的，在一定条件下也可以实用，但它距离完全实用还有些差距。例如，我们只用它传送字符文件，限制传送的文件长度不大于20000个字节等等。

这个程序用到三个外部过程，是用汇编语言实现的。其中，Init过程完成对接口按给定的参数进行初始化。Send过程是执行从发送一方发送一个字符，而Receive过程是接收一方执行接收一个字符的功能。

主程序很简单。它要在发送与接收方同时运行。首先在屏幕上显示问题：本机是接收方还是发送方。并问读写（被传送的）的文件的文件名，得到正确回答后，发送与接收方都要对串行口进行初始化操作，并且，发送方打开被传送的文件，逐个字符（包括回车进行控制符在内）用串行口向外发送，并同时将其显示在终端上。发送的最后送一个特定字符'|'表示结束。而接收一方则新建一个文件，并接收发送一方送来的每一个字符，先存入数组ar中。遇到特定字符'|'后停止接收，随后将ar中的内容写到文件F中。

最后，发送与接收方分别关闭文件。

该程序开始运行时，应先在接收一方回答完毕，使其处于等待发送一方发送字符的状态，然后再结束发送一方的最后回答。

下面是该程序的源清单，和它用到的三个外部过程的汇编语言程序的源清单。

```
program x (input, output);
var f :file of char;
    fnam:lstring (20);
    ch:char;    i, j: integer;
    ar: packed array [1..20000] of char;

    procedure init; extern;
    procedure send (a:char); extern;
    pocedure receive (var ch:char); extern;

begin
    writeln; writeln;
    write ('receive(r)or send(s)? '); readln(ch);
    write ('please give file name:'); readln(fnam);
    init;    assign (f, fnam);
    if (ch='S') or (ch='s')
        then [reset (f);
                while not eof (f) do
                    [ch:=f^; send(ch); write(ch); get(f)];
                send ('|');
            ]
        else [rewrite (f);
```

```

    receive(ch), i:=0,
    while ch< >'| 'do
        [i:=i+1, ar[i]:=ch, receive(ch)] ;
    for j:=1 to i do write (f, ar[j]) ;
close(f)
end.
codesg segment
    assume cs:codesg, ds:codesg
    public init, send, receive

init    proc    far
        mov     dx,     3fbh    ,
        mov     al,     80h
        out     dx,     al
        mov     dx,     3f8h
        mov     al,     60h
        out     dx,     al
        mov     dx,     3f9h
        mov     al,     0
        out     dx,     al
        mov     dx,     3fbh,
        mov     al,     0ah    ,
        out     dx,     al
        mov     dx,     3f9h
        mov     al,     0
        out     dx,     al    ,
        ret

init    endp
send    proc    far
        push    bp
        mov     bp,     sp
        mov     ax,     [bp+6]
        push    ax
        xchg    al,     ah
        pop     ax
        cmp     al,     '| '
        je      exitl
        call    sendchr    , send out a char in AL
        pop     bp
        ret     2

```

```

exit1:  mov     al,      ' '
        call   sendchr
        pop     bp
        ret     2
send    endp

sendchar proc
        push    ax
        mov     dx,      3fch
        mov     al,      3
        out     dx,      al
        mov     dx,      3fdh
send2:  in      al,      dx
        test    al,      20h
        jz      send2
        pop     ax
        mov     dx,      3f8h
        out     dx,      al
        ret
sendchr endp
receive proc far
        push    bp
        mov     bp,      sp
        push    ax
        mov     dx,      3fch
        mov     al,      1
        out     dx,      al
receive1: mov     dx,      3fdh
        in      al,      dx
        test    al,      1
        jz      receive1
        mov     dx,      3f8h
        in      al,      dx
        mov     si,      [bp+6]
        mov     [si],    al
        pop     ax
        pop     bp
        ret     2
receive endp
codesg  ends
end

```

10.5 在PASCAL程序中读写dBASE的文件

许多PC机用户，都用dBASEⅢ建起了功能不等的各种数据库管理系统，计算机系统中已经保存着大量的有用数据。能否在PASCAL程序中使用、维护这些已有数据，是这些用户很关心的一件事，是他们愿不愿意使用PASCAL语言时必然要想到的。我们的回答是肯定的，而且可以说，在PASCAL的程序中读写dBASE的文件很容易实现。

下面先简单介绍dBASEⅢ的数据文件的结构。在PC机系统中，这类文件的扩展名均为.dbf，是用dBASEⅢ的CREATE命令建立起来的。数据文件可以分成两部分：

(1) 结构描述部分，分为两个表：

第一个表，由文件的前32个字节组成，是结构描述的固定部分，具体内容安排如下：

字节序号	长 度	内 容
0	1	dBASE的版本号
1 ~ 3	3	最后修改日期的年、月、日
4 ~ 7	4	文件中已有记录数
8 ~ 9	2	数据首记录地址
10 ~ 11	2	记录长度
12 ~ 31	20	保留备用

第二个表，从文件的第32个字节开始，以后每32个字节组成一个域的定义，域定义的数量取决于每个关系的域数，这是结构描述的变化部分。每一个域的定义的格式是：

字节序号	长 度	内 容
0 ~ 10	11	域名
11	1	域类型（用ASCII码字符表示）
12 ~ 15	4	域数据地址
16	1	域长度
17	1	小数位数
18 ~ 31	14	保留备用

域定义之后用2个字节作为全部域定义的结束标记，其内容为#0D00，#0D是回车符的ASCII码值。

(2) 数据记录的存放区，用ASCII字符形式顺序地存放数据的记录内容。

上述内容是通过分析、检查用DOS系统下debug命令卸出几个dBASEⅢ数据文件的内容了解到的。我们在下面给出了卸出的student.dbf文件的完整内容。每一行的最左部分，是文件内容读进内存后，16个字节在内存中的首地址，中间部分是用16进制给出每个字节的具体内容，右边部分用字符形式给出每个字节中的内容，“.”也被用来表示不可打印的字符编码。字节的值小于16进制的20，就属于不可打印的字符编码。

```

C:\>debug
--n student.dbf
-l
-d
1207:0100  03 50 01 01 04 60 00 00-E2 00 22 00 00 00 00 00 .P ..... " .....
1207:0110  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
1207:0120  4E 41 4D 45 00 00 00 00-00 00 00 43 07 00 08 3E  NAME...C.....>
1207:0130  0A 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
1207:0140  41 52 41 44 45 00 00 00-00 00 00 43 11 00 08 3E  GRADE...C.....>
1207:0150  02 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
1207:0160  42 49 52 54 48 59 45 41-52 00 00 4E 13 00 08 3E  BIRTHYEAR..N>
1207:0170  08 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
-d
1207:0180  50 4F 49 4E 54 53 00 00-00 00 00 4E 1B 00 08 3E  PA...F...N.....>
1207:0190  06 01 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
1207:01A0  4D 41 54 48 00 00 00 00-00 00 00 4E 21 00 08 3E  MATRI..N1...>
1207:01B0  06 01 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
1207:01C0  42 4E 59 00 00 00 00 00-00 00 00 4C 27 00 08 3E  BOY...L'.....>
1207:01D0  01 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
1207:01E0  0D 00 20 4C 49 20 4D 49-4E 47 20 20 20 32 20 20...LI MING  2
1207:01F0  20 20 20 31 39 37 30 20-35 37 36 2E 30 20 31 39      1970 753.0  10
-d
1207:0200  30 2E 30 59 20 57 41 4E-47 20 59 49 4E 47 20 31 0.6Y  WANG
                                           YING 4
1207:0210  20 20 20 20 20 31 39 30-38 20 36 31 30 2E 30 20      1968 610.0
1207:0220  20 39 37 2E 35 4E 2A 48-55 41 4E 47 20 59 49 20 97.5N*HUANGYI
1207:0230  20 31 20 20 20 20 20 31-39 37 31 20 35 36 34 2E 1      1971 564.
1207:0240  30 20 20 39 39 2E 30 59-20 5A 48 41 4E 47 20 48 0 99.0Y ZHANGH
1207:0250  4F 55 20 32 20 20 20 20-20 31 39 37 30 20 34 39 OU 2      1970 49
1207:0260  39 2E 35 20 20 35 32 2E-35 4E 1A 8B 46 F0 89 46 9.5 52.5N..F..F
1207:0270  F4 8B 46 0A 03 46 0C 89-46 EA 83 7E EA 01 7C 4F...F..F...F.....io

```

前两行的32个字节，是文件结构描述的固定部分，具体内容表明，dBASE的版本号为Ⅱ，文件建立或修改的最后日期为80年1月1日，文件中已有4个记录，其首记录地址为E2，记录长度为31。接下来的12行为该文件结构描述的变化部分，其内容为该文件记录的六个域的具体定义。我们可以用dBASE的命令dilis tstru检查这部分内容，具体结果如下：

```

. use student
. dilist stru
Structure for database:A:student.dbf
Number of data record, :      4
Date of last update      :01/01/80

```

Field	Field name	Type	width	Dec
1	NAME	Character	10	
2	GRADE	Character	2	
3	BIRTHYEAR	Numeric	8	
4	POINTS	Numeric	6	1
5	MATH	Numeric	6	1
6	BOY	Logical	1	
* * Total * *			34	

可以把这里得到的内容与前面卸出的内容加以比较, 验证二者间的对应关系。

在以内容为 #0D 和 #00 两个字节结束域定义之后, 就开始了数据记录的具体内容, 都是用字符串形式表示的。第一条记录从 E2 地址开始, 以后的每 34 个字节存放一条记录, 其内容可用 dBASE 的命令 `ldisplay all` 加以显示, 结果为:

```
. ldisplay all
Record#   NAME      GRADE BIRTHYEAR POINTS MATH  BOY
1        LI MING      2      1970      576.0  100.0  .T.
2        WANG YING    4      1968      610.0   97.5  .F.
3 *      HUANG YI     1      1971      564.0   99.0  .T.
4        ZHANG HOU    2      1970      499.5   52.5  .F.
. quit
* * * END RUN dBASE █
```

其中第 3 条记录在姓名前有个 “*”, 表示该条记录已被删除。

有了上述知识, 用 PASCAL 语言写出读取 dBASE 的数据文件的程序就很方便了。我们给出了读取 `student.dbf` 文件的程序例子。该程序主要用来表明读取 dBASE 文件的原理性方法, 离实用标准还有一定差距, 但改成完全实用的程序并不很难了。

在这个程序中, 把文件结构描述的固定部分说明为 `string (32)` 类型的变量 `STR`, 并通过地址类型变量 `ADRSTRUC` 来引用结构描述中的每一个域, 包括 dBASE 的版本号, 文件建立或修改的最后日期年、月、日, 文件中已有的记录数, 数据首记录的地址和记录长度等。

程序中, 把文件结构的描述的变化部分, 即全体域的定义, 说明为由 30 个元素组成的数组。每个数组元素描述一个域, 同样也被说明为类型 `STRING (32)` 和类型 `FIELD`, 并通过地址类型变量来引用域定义中的每一个数据项。并在 `FIELD` 定义中明确给出前三个域的具体起始偏移值。

对文件的数据记录的结构, 是用类型 `REC DATA` 来描述的, 该记录中的每一个域的偏移值, 都在定义中明确给出了, 这是正确使用数据记录所必定要求的。这些偏移值可以在文件结构的域定义部分中找到。这里也是用 `STRING (34)` 和 `REC DATA` 两种类型来描述每一个数据记录的, 并用地址类型变量来引用类据记录的每一个域值 (都是 `STRING` 类型)。

在读 `student.dbf` 文件时, 总是把文件中的每个字符读到相应 `STRING` 类型的变量中, 而在显示读来的内容时, 都是通过地址类型的变量直接引用相应的域值的。

下面简单地解释一下程序中的有关语句。

程序的前两行是读取文件名并打开文件。接下来的三行是确定地址类型变量的初值。

再接下来的一行，用于读取结构描述的固定部分。由此，就得到了文件中已有数据的记录数目和数据首记录的起始地址，并由此可以计算出组成每条记录的域数。

接着的三行实现计算域数，并用FOR语句读取全部的域定义。

用两个GET(F)跳过域定义结束标记后，接下来的三行读取文件中已有的全部记录。

最后的三小段程序分别用于输出文件结构描述的固定部分，全部的域定义信息和数据记录的实际内容。在这三小段程序中，都是用地地址类型变量，通过相应定义中给出的域名引用相应的域值的。

通过给在程序清单之后的运行结果，可以验证程序运行的正确性。可以把这里给出的结果与用dBASE命令得到的结果进行比较，会看到一些微小差别，但都不是实质性的。如第三条记录内容带有删除标记，没输出，男女(BOY域)一项直接输出字符'Y'或'N'。域定义中的域的数据类型直接输出字符'C'、'N'或'L'等。

下面给出的是程序的源清单和程序的运行结果。

```
program stru (input, output) ;
type
  structure=record
      vers, year, month, day:byte;
      recnumb:integer4;
      firstrec, reclen:integer;
  end;
  field=record
      name      [00] :string (11) ;
      fldtype   [11] :char;
      dataadr   [12] :integer4;
      len, dec:byte;
      reserved:string (14)
  end;
  str32=string (32) ;
  fldtab=array [1..30] of string (32) ;
  fldary=array [1..30] of field;
  recdata=record
      flag      [00] :CHAR;
      name      [01] :string(11) ;
      grade     [11] :string (2) ;
      birthyear [13] :string (8) ;
      points    [21] :string (6) ;
      math      [27] :string (6) ;
      boy       [33] :char
  end;
  datastr=array [1..50] of string (34) ;
var
```



```

f:text;                                fname:string (12) ;
i, j, k:integer;                       str:str32;
adrstruc:adr of structure;              adrstr:adr of str32;
adrfld:adr of fldary;                   adrarr:adr of fldtab;
adrdatarec:adr of datastr;              adrrec:adr of recdata;

fld:fldtab;                             fldnumb, recnumb:integer;
datarec datastr;

begin
  write ('file name:');                  readln (lname) ;
  assign (f, fname) ;                   reset (f) ;
  adrstr:=adr str;                       adrstruc.r:=adrstr.r;
  adrarr:=adr fld;                       adrfld.r:=adrarr.r;
  adrdatarec:=adr datarec;               adrrec.r:=adrdatarec.r;

  str [1] :=f^; for i:=2 to 32 do [get (f) , str [i] :=f^] ;

  fldnumb:=adrstruc^.firstrec div 32-1;
  for k:=1 to fldnumb do
    for i:=1 to 32 do [get (f) , fld [k, i] :=f^] ;

    get (f) ,      get (f) ;
    if adrstruc^.recnumb<32767
      then recnumb:=ord (adrstruc^. recnumb);
    for i:=1 to recnumb do
      for k:=1 to 34 do [get (f) ,      datarec [i, k] :=f^] ;
  writeln ('structure for database:', lname) ;
  writeln ('Number of data records:', adrstruc^.recnumb:5) ;
  writeln ('Data of last update      ', adrstruc^.day:2, '/',
          adrstruc^.month:2, '/', adrstruc^.year:2) ;

  writeln ('Firstrecord address  ', adrstruc^.firstrec:5) ;
  writeln ('Record length        ', adrstruc^.reclen:5) ;
  writeln;

  writeln ('Field number:', fldnumb:4) ;
  writeln ('Field Field name Type Width Dec') ;
  for j:=1 to fldnumb do with adrfld^ [j] do
    [write (j:4, name:16, fldtype:3, length:8) ;

```

```

        if dec<>0 then writeln (dec:8) else writeln] ;
writeln;
writeln ('Data record contents:');
write ('Rcord#');
for j:=1 to fldnumb do write (adrfld^[j] .name);
writeln;
for j:=1 to recnumb do with adrrec^ do
    [if flag<>'*' then
        writeln (j:4, ' ', name:11, grade:6, birthyear:13,
            points:11, math:10, boy:8);
        adrrec.r:=adrrec.r+34];
close (f)
end.

```

程序的运行结果如下:

file name:student. dbf

Structure for database :student. dbf

Number of data records : 4

Data of last update : 1/1/80

Firstrecord address : 226

Record length : 34

Field number:6

Field	Field name	Type	Width	Dec
1	NAME	C	10	
2	GRADE	C	2	
3	BIRTHYEAR	N	8	
4	POINTS	N	6	1
5	MATH	N	6	1
6	BOY	L	1	

Data record contents:

Rcord#	NAME	GRADE	BIRTHYEAR	POINTS	MATH	BOY
1	LI MING	2	1970	576.0	100.0	Y
2	WANG YING	4	1968	610.0	97.5	N
4	ZHANG HOU	2	1970	99.5	52.5	N

这个程序存在的一个主要问题是通用性差, 是针对student.dbf这个具体文件而设计的, 它不能用于读其它不同结构的文件。在实际工作中, 很难设想为每个不同结构的文件写一个专用的读写程序, 因此, 实现一个通用的、能用于读dBASE中任何一个数据文件的程序模块是需要的, 此时必须解决以下几个问题,

(1) 文件结构中域定义信息的管理。我们在前面的程序中,是用30个域定义组成的数组表示的。为了能管理任意个数的域定义,可以把该数组的下标范围扩大,并取消每一域定义中暂时未用的14个字符,以减少域定义信息占用的内存空间,或把域定义说明为高级数组类型,在程序中按实有域数动态生成所需大小的数组,例如:

```
FLDTAB=SUPER ARRAY [1..*] OF STRING (18),  
P: ↑FLDTAB,  
NEW (P, FLDNUMB)
```

(2) 文件中数据记录数可能很多,很可能没法全部读入内存,此时,既可以用一次读入一条数据记录的顺序处理方式,也可以读入规定数目的记录到程序的数组中,该数组的大小可以按机器可用内存空间、程序本身长度和程序运行效率等多项因素综合考虑。

(3) 数据一条记录中各域的域名、长度、起始地址等都可以在域定义中找到,但如果不采用预编译技术,我们还是无法直接给出一条记录的准确定义。如前面程序中的RECD-ATA类型,是在我们已经知道了数据每条记录的域名和长度之后才写的程序,可以在程序中准确地给出它的定义。要实现通用程序模块,域名、域数据类型和域长度,都要到域定义表中去查找,并用定义子串、分解串和拼接串等技术处理每一条记录中的不同域。在对数值域(类型为N)进行处理时,小数点位置DEC为0,代表整型量,DEC不为零,代表实型量,DEC的值表示小数点之后要保留的十进制的有效数字的位数。要把用字符串表示的数字变成二进制形式的数值,必须调用DECODE函数完成。

(4) 文件中已有数据记录数是用INTEGER4形式表示的,这种类型的数据既不能用作FOR语句的循环控制变量,也不能用于数组的下标。当该值大于32767时,要注意变它为两个整型量相乘的形式,或用WHILE语句、REPEAT语句使用它的值。

(5) 数据文件本身是顺序文件,由前顺序向后读是方便的,当读到后面某条记录后再想读前边的记录就麻烦了。为此,在需要时,可以通过计算该记录的地址并通过按磁盘块号读的方式等办法来提高文件读取的效率,或通过建立临时的直接读写文件等方式来处理。在此,我们无意深入讨论这些问题。

以上的五个问题,以(3)和(5)比较复杂,但设计一些辅助过程后,还是可以达到简化其使用难度和提高程序运行效率两项目标的。

顺便指出,dBASE建立的索引文件,也是可以在PASCAL程序中直接使用的。我们还可以用PASCAL程序建立dBASE能直接使用的数据文件和索引文件。这些问题要更复杂一点,限于篇幅,就不介绍了,请自己摸索解决。

附录A ASCII字符集

在计算机程序设计语言中，目前采用得最普遍的字符编码方案是 ASCII(American Standard Code for Information Interchange) 方案。它包含 94 个可打印的字符和 1 个空格字符（编码为 10 进制的 32）。此外还有 33 个控制字符，编码为 00 至 31 和 127。MS PASCAL 把编码 128 至 255 的 128 个符号也看作为字符。下表给出的是 94 个可打印字符的编码值。

ASCII 字符编码表

		低 位 数 字									
		0	1	2	3	4	5	6	7	8	9
高 位 数 字	3				!	"	#	\$	%	&	'
	4	()	*	+	,	-	.	/	0	1
	5	2	3	4	5	6	7	8	9	:	;
	6	<	=	>	?	@	A	B	C	D	E
	7	F	G	H	I	J	K	L	M	N	O
	8	P	Q	R	S	T	U	V	W	X	Y
	9	Z	[\]	^	_	`	a	b	c
	10	d	e	f	g	h	i	j	k	l	m
	11	n	o	p	q	r	s	t	u	v	w
	12	x	y	z	{		}	~			

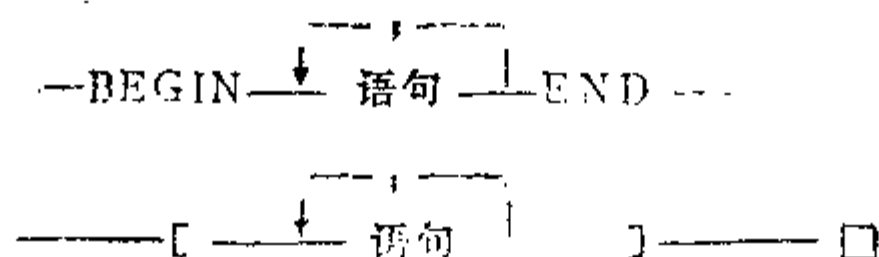
附录B MS PASCAL语言的语法图

这里给出的是MS PASCAL语言语法的形式描述,采用的是简易的语法图,并且是按实现编译程序时经常用到的递归下降法来进行这种描述的。

语法图,是描述计算机程序设计语言语法规则的一种常用手段,它简洁清晰,直观易懂,特别是对广大非计算机专业的人员,更显得易学好用。下面我们首先把怎样看懂和查询语法图的简单知识讲解清楚。会用语法图,对学好用好一种程序设计语言是很有帮助的。

先给出对复合语句的语法描述的例子,并通过这个例子解释语法图的构成方法和它的含义。

复合语句:



在这条描述中,用到了以下几种符号:

(1) 中文单词,如复合语句、语句。在这里它们都代表一个语法范畴,都需要有一条用来描述它们的语法图。这里的复合语句是被描述的一个语法范畴,而语句则是描述复合语句时用到的语法范畴,还一定要有一条描述语句的语法图。一个语法范畴,或称语法概念,代表的是一类(而不是一个)符号,如这里说的复合语句就代表全部可能用到的复合语句。所谓语法描述,指的是给出被描述的语法范畴的构成规则。

(2) 英文单词,如这里的BEGIN和END。它们是该语言的保留字或预说明的标识符,它们代替特定的含义,不需要再描述。在用户程序中,它们必须按语法图中的位置和情形来使用。

(3) 专用字符,如这里的[;]。它们也一定要按语法图中给出的位置和情形来使用。

(4) 符号“□”不是PASCAL语言使用的一个符号,它只用来表示一条语法图的结束。

(5) 实线,表明各符号间的次序关系。处于并列位置的实线表示可以取其中任何一条。

(6) 虚线,代表可重复多遍的顺序关系,所以它要求必须有和它并列的实线给出重复的内容,它还要有箭头表明重复的次序关系。

在进行了简单的上述说明后,我们可以把上面给出的语法图的含义叙述如下:

(1) 复合语句是或者由关键字BEGIN和END“夹”起来的一个、多个语句组成,或者由左方括号和右方括号“夹”起来的一个、多个语句组成。在被夹起来的是多个语句的情况下,各语句之间通过分号隔开。

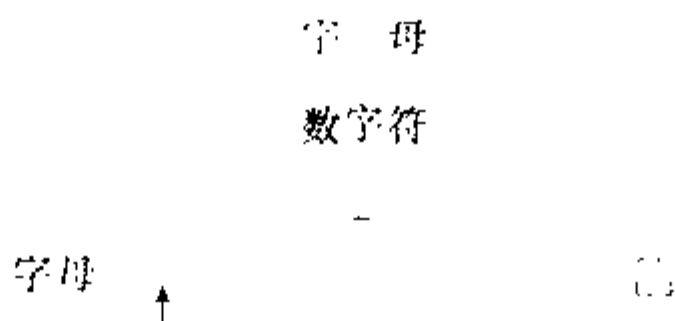
(2) 保留字BEGIN或左方括号与第一个语句之间、END或右方括号与最后一个语句之间不必有分号。

(3) 复合语句只能这样组成, 满足这种规定的就是合法的复合语句, 否则一定是不正确的复合语句, 或者根本不是复合语句。

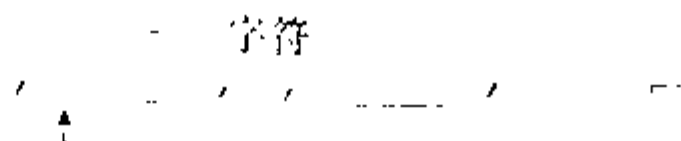
在这里给出的完整的语法图中, 字母是指大写和小写的各26个英文字母。数字符通常是指表示10进制的数用到的0到9这十个阿拉伯数字, 但当用二进制表示一个整型数值或字类型的数值时, 只能使用0和1这两个数字符。类似的, 用八进制时, 只能使用0到7这八个数字符, 用十六进制时, 除了能用0到9这十个数字符外, 还可以用大写或小写的英文字母A到F(或a到f)来表示10到15这六个数值。在提到字符时, 通常情况下是指95个可打印的字符, 但MS PASCAL语言把编码值0到255这256个字符都包括了进去, 它是不包括单引号“'”这个字符, 这是为了谈论与识别字符串更方便些。

我们建议那些不大熟悉语法图的读者, 花一点时间学懂它, 在学习PASCAL语言程序设计的过程中一定会收到事半功倍的效果。

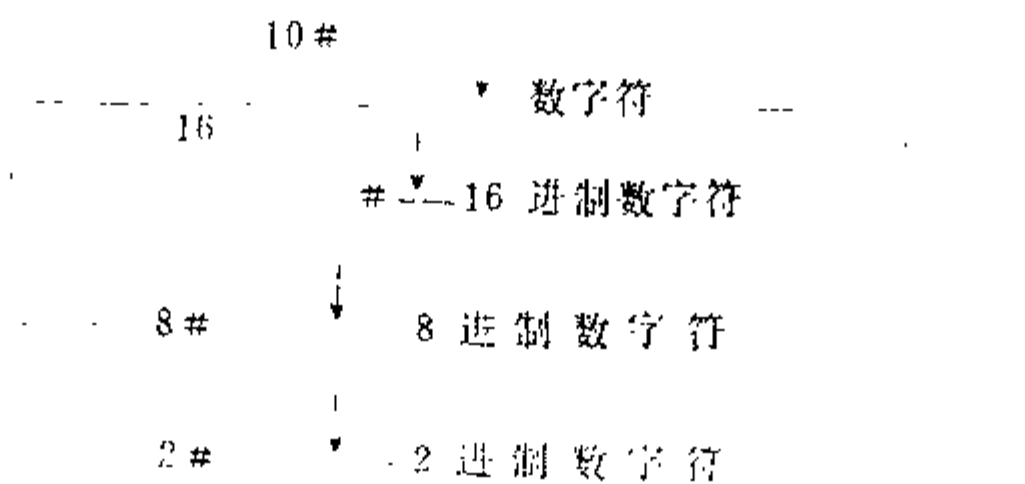
标识符:



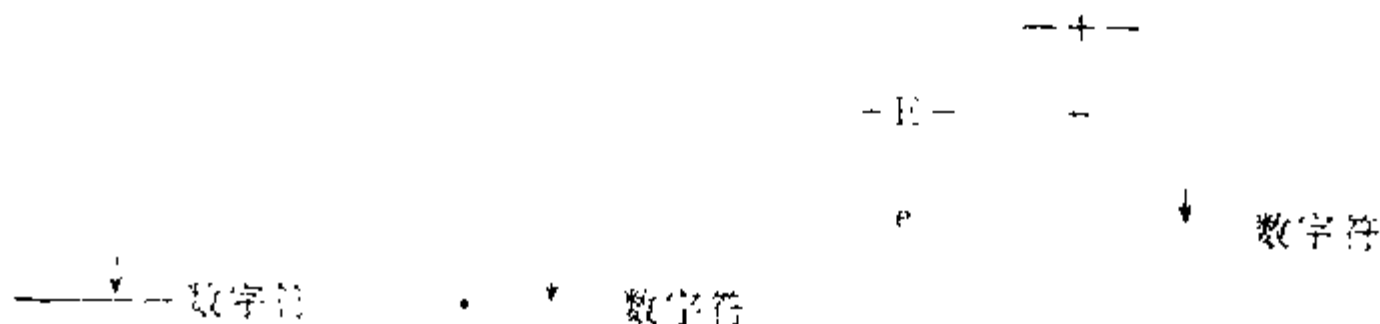
字符串, 文字说明:



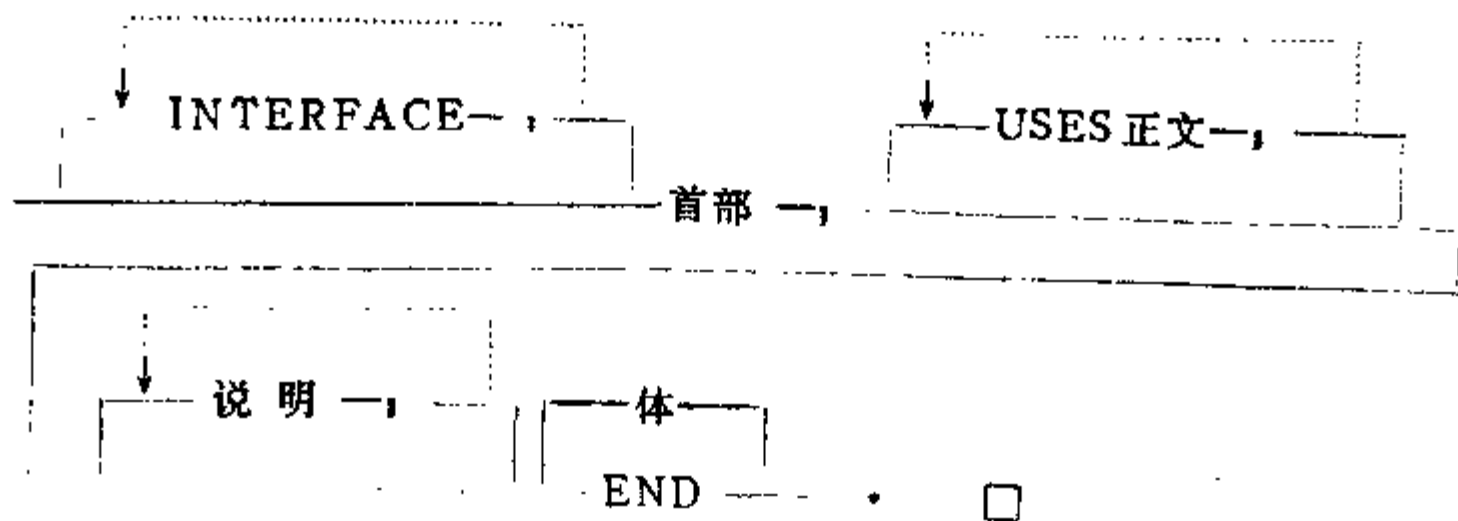
整型数:



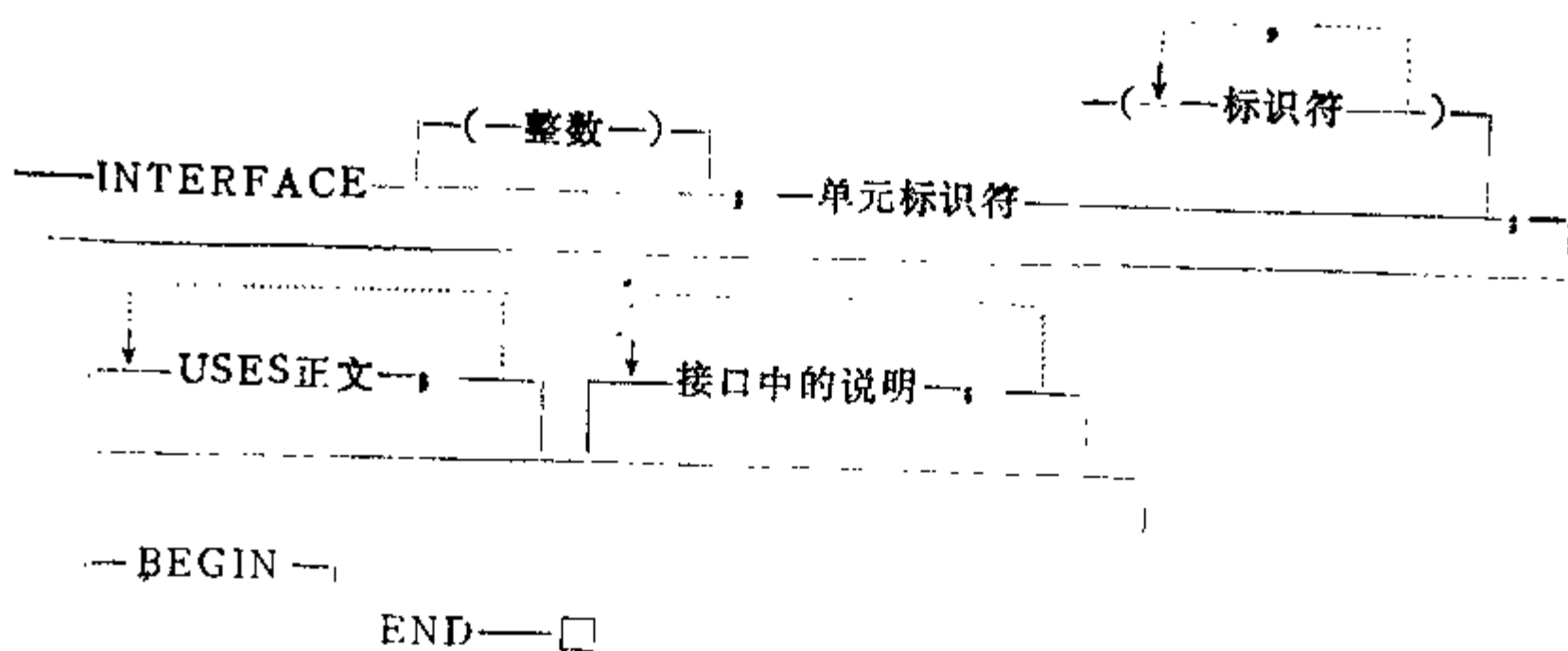
实型数:



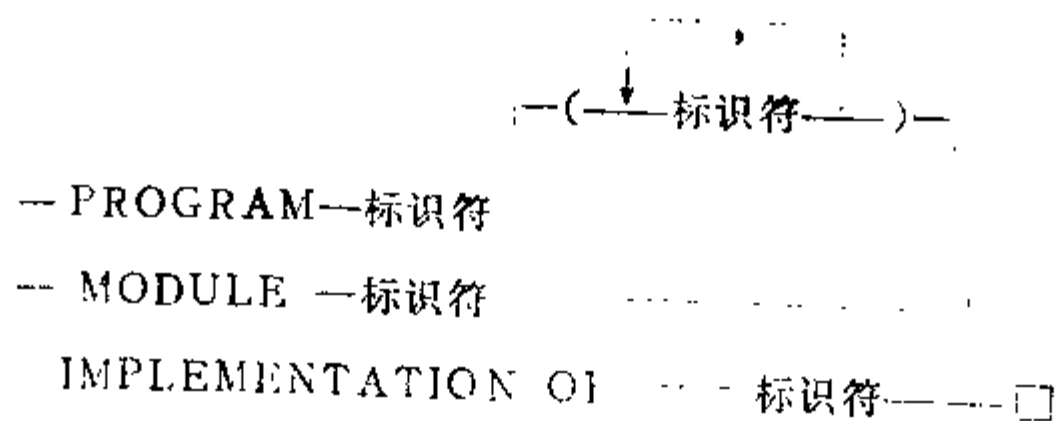
编译对象:



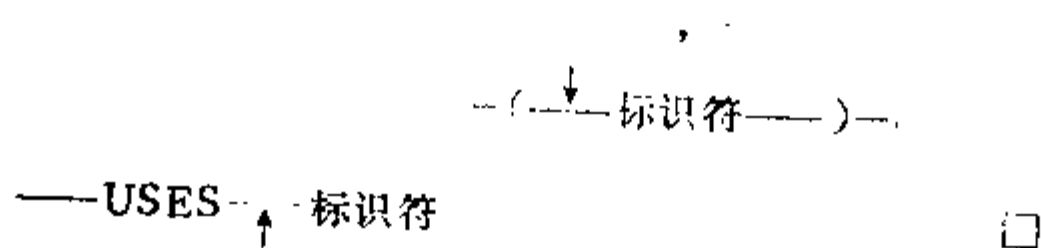
接口:



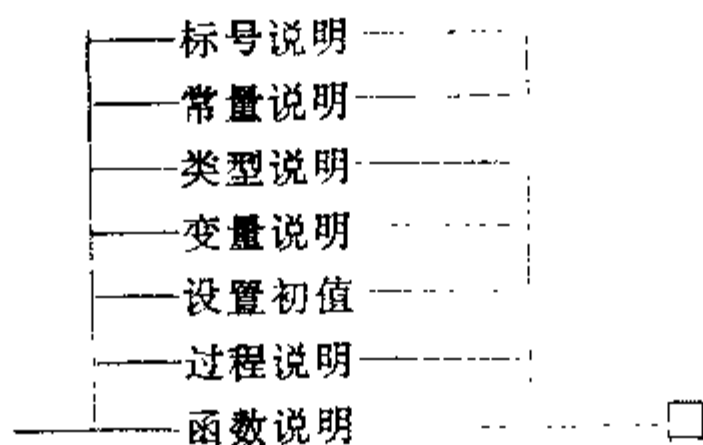
首部:



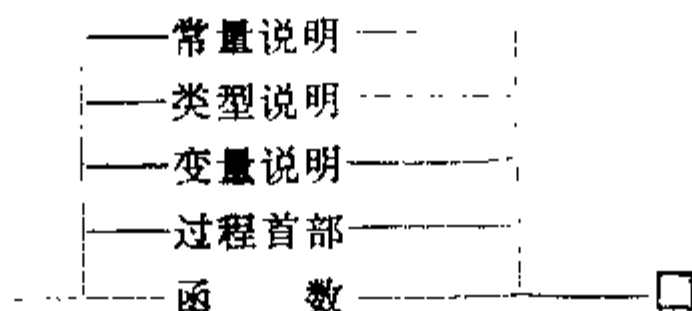
USES正文:



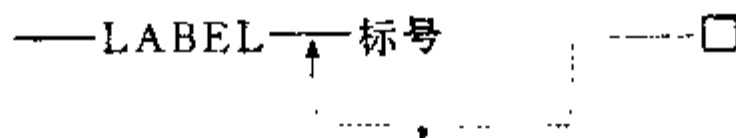
说明:



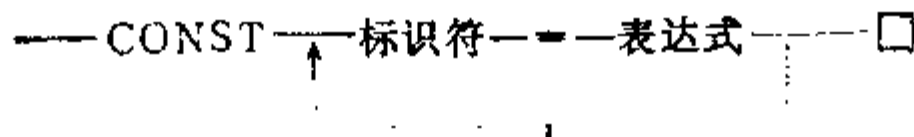
接口中的说明:



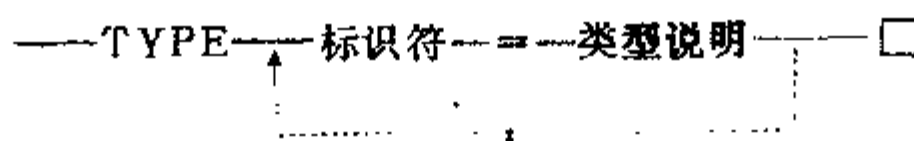
标号说明:



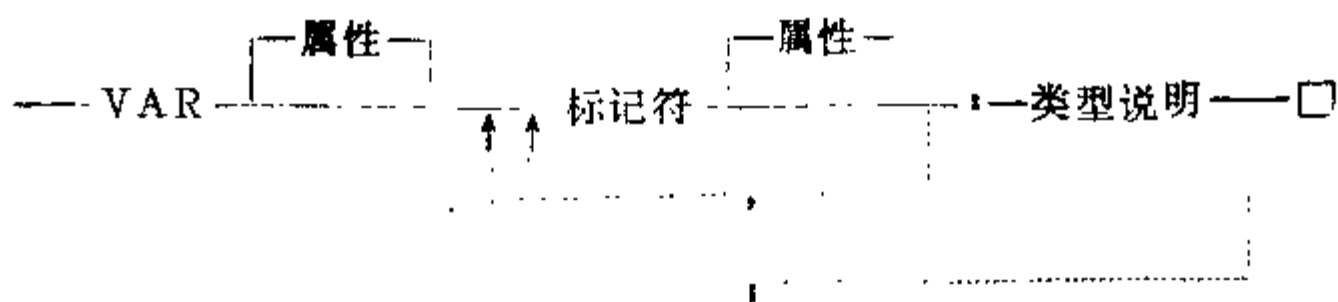
常量说明:



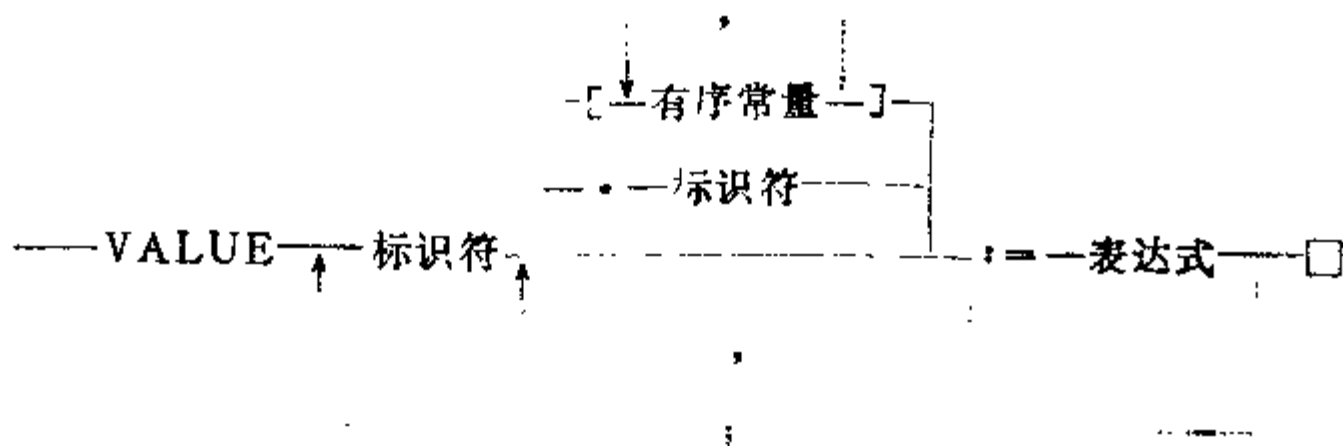
类型说明部分:



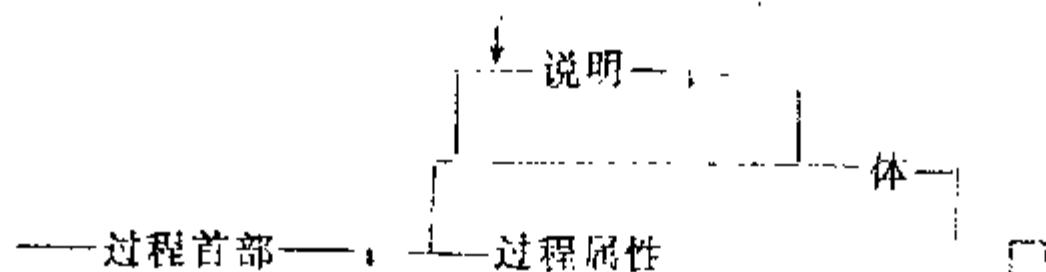
变量说明部分:



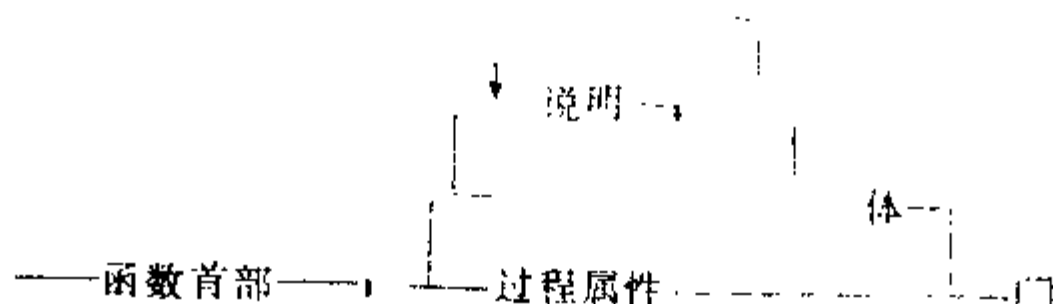
设置初值:



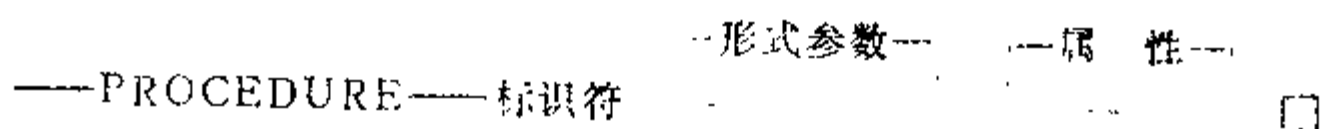
过程说明:



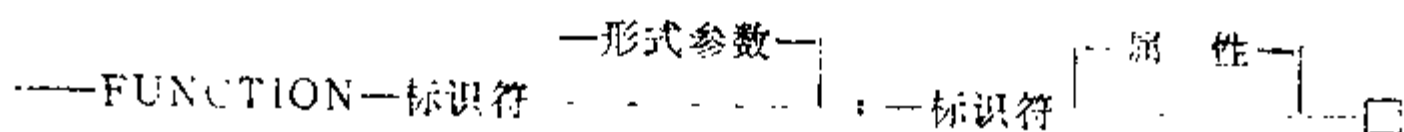
函数说明:



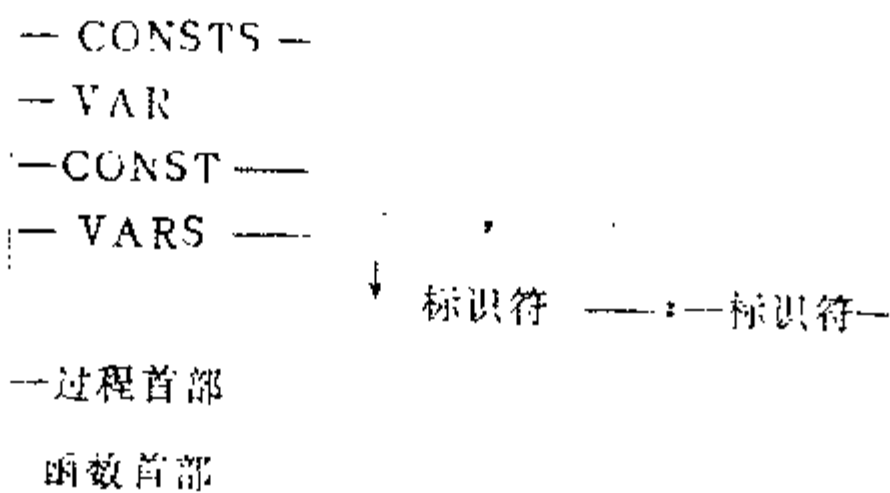
过程首部:



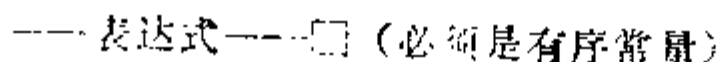
函数:



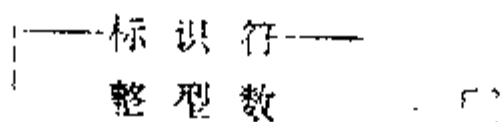
形式参数:



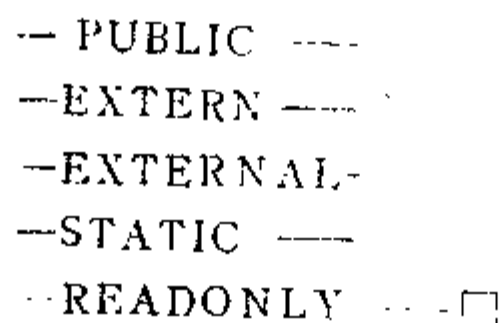
有序常量:



标号:



变量属性:



过程属性:

—FORWARD—
—EXTERN—
—EXTERNAL— □

类型说明:

— ^
—ADR OF—
—ADS OF—
—
—
— (—整型数—) —
—标识符—
—有序类型—
—SUPER— —PACKED— —有序类型
—ARRAY— [—有序常量—... *
—域— —]—OF—类型说明—
—RECORD — END
—PACKED— —SET OF— 有序类型 —
—FILE OF— 类型说明 —

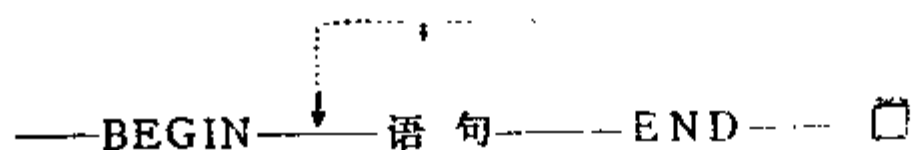
有序类型:

—标识符—
— (—标识符—) —
有序常量—...—有序常量

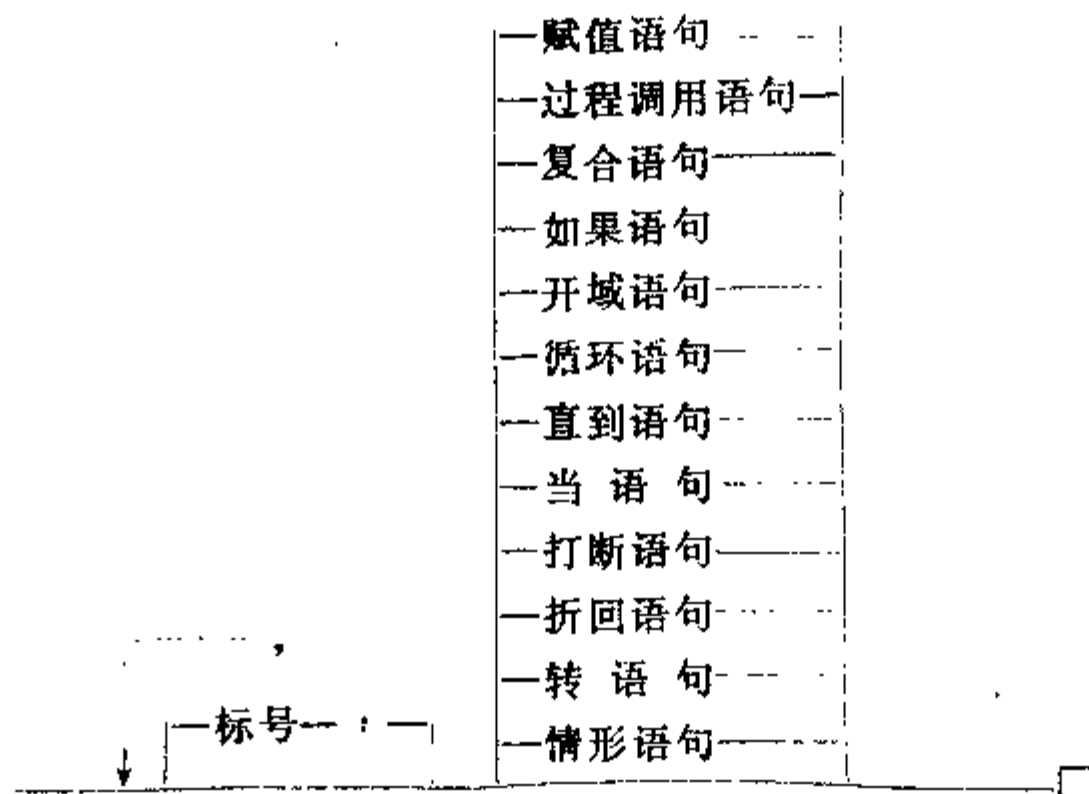
域:

— [—有序常量—] —
—标识符— :—类型说明
— [—有序常量—] —
—标识符—
CASE- —标识符— —标识符—
—OF— —有序常量—
— (—域—) —

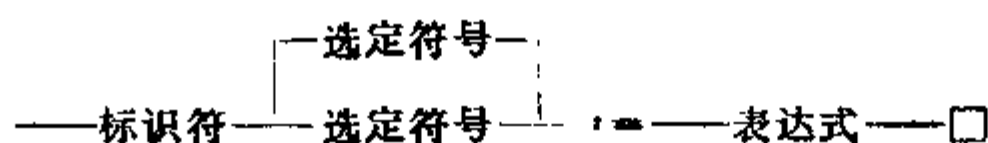
体:



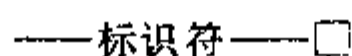
语句:



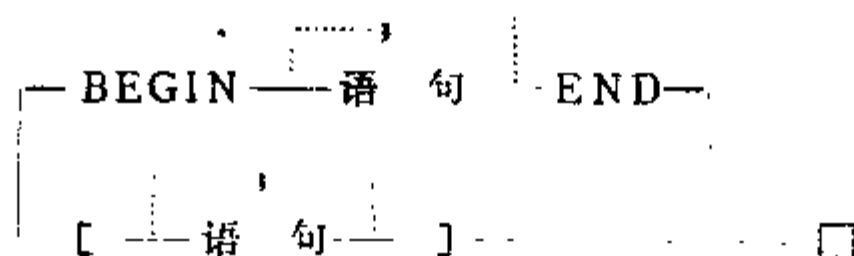
赋值语句:



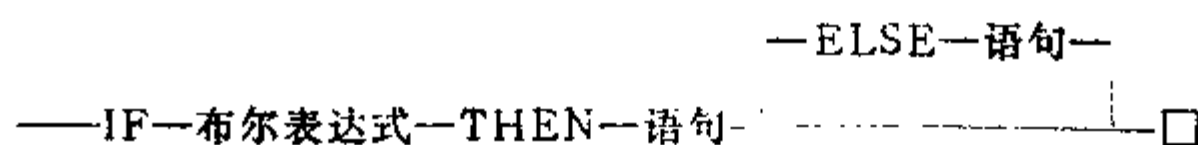
过程调用语句:



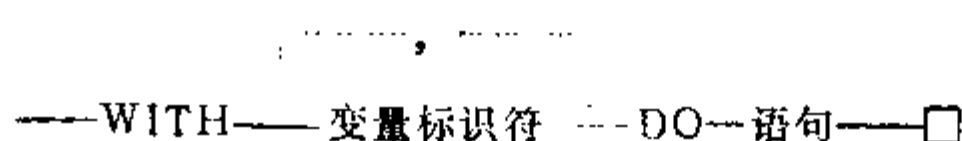
复合语句:



如果语句:



开域语句:



循环语句:

——FOR——
——STATIC——
——标识符——:——表达式——TO——
——表达式——DOWNTO——
——表达式——DO——语句——□

直到语句:

——REPEAT——语句——UNTIL——布尔表达式——□

当语句:

——WHILE——布尔表达式——DO——语句——□

折回语句:

——CYCLE——
——标号——□

打断语句:

——BREAK——
——标号——□

转语句:

——GOTO——标号——□

情形语句:

——CASE——有序表达式——OF——
——有序常量——
——有序常量——
——有序常量——
——语句——
——OTHERWISE——
——语句——
END——□

有序表达式:

——表达式——□ (必须是有序表达式常量)

布尔表达式:

——AND THEN——

——OR ELSE——
——表达式——

——表达式——□

选定符号:

，
--[—— 有序表达式——]
- · - - 标识符 - - -
^ □

表达式

--<-----
---<=---
-->=---
· > ---
· = - - -
--<>--
-- IN --简单表达式--
--简单表达式-----

简单表达式:

-----+-----

- OR -
+ - - -
- - - -
XOR · · 项
项 项

项:

· - * - -
-----/
--DIV--
-- MOD --
AND 因子
- 因子 -

因子:

—标识符——

—前数引用— IDENTIFIER ACTUALS

—字符常量— —选定符号—

—结构常量—

—标识符——

集合常量

(SET CONSTANT)

整型数

实型数

(— 表达式 —)

NOT 因子

NIL

ADR—标识符

ADS—标识符

附录C 出错信息一览表

编译程序和运行系统都能发现用户程序中的一些错误。编译程序由PAS1.EXE和PAS2.EXE两部分组成。PAS1.EXE完成对用户程序的语法和语义检查,并要生成中间代码,它能发现用户程序中大部分错误。PAS2.EXE完成代码优化和代码生成,很少遇到用户程序出错的情况。运行系统检查出的错误被分为两大类:文件系统运行错误和其它错误。下面就按这四种情况给出出错信息一览表。

1. PAS1.EXE发现的错误

这类错误或警告信息,都是由一个错误编号和一项错误简要说明两部分内容组成的。这两部分内容,以及一串破折号和一个表示出错位置的箭头,都将出现在编译清单文件或终端屏幕上。它们中的大多数,要出现在出错语句的下一行,只有编号为128、129和130三项出错信息出现在\$SYMTAB列表内容的后面。

这里的错误被分为三类:警告性错误,实际错误和致命错误。警告性错误,是指编译程序已经发现了用户程序的错误,但是它能按确定的规则“应付”这个错误并继续生成中间文件,有可能得到正确的运行结果,但并无确实把握。对这类错误,是用在出错信息前加上“Warning”这个英文单词来区分的。实际错误,是指编译程序发现了用户程序中的错误,并且无法判断应如何继续生成相应的中间文件,只能停止生成中间文件,只对用户程序的后面部分进行语法和语义检查,因此编译之后得不到中间文件。致命错误,是指发现了用户程序中的一项错误,它使编译程序无法继续编译,在出错信息中还将给出如下信息:

Compiler cannot Continue (无法继续编译,编译程序将停止编译。

造成致命错误的条件包括:

- 错误太多,超过了用\$ERROR规定的最大值
- 用户源程序异常结束
- 标识符的辖域嵌套太深
- 找不到关键字PROGRAM、MODULE或IMPLEMENTATION
- 找不到PROGRAM、MODULE或IMPLEMENTATION的名字。

错误信息“Compiler”,指的是编译过程中进行内部一致性检查失败,属于编译程序本身出错,极少遇到这种情况。

错误清单

- 101 Invalid Line Number (无效行号)
行号大于32767,源程序的行数太多
- 102 Line Too Long Truncated (行太长,截尾)
源程序的行长限制在142个字符之内
- 103 Identifier Too Long Truncated (标识符太长,截尾)
长子最大限度的标识符要截尾。
- 104 Number Too Long Truncated (数字太长,截尾)

- 数值常量的长度限制于标识符的长度限制相同
- 105 EHD Of String Not Found (没有找到字符串的结尾)
在找到串的闭合引号之前, 该行已经结束。
- 106 Assumed String (认为是字符串)
采用了双引号来表示字符串, 应使用单引号。
- 107 Unexpected End Of File (非预期的文件结束)
编译扫描时, 遇到用数, 元命令等等来结束文件。
- 108 Meta Command Expected Command Ignored
(元命令不完整, 不能处理)
在注释的开头, \$ 之后没有元命令标识符
- 109 Unknown Meta Command Ignored (忽略未知的元命令)
元命令的标识符是未知的, 或是无效的。
- 110 Constant Identifier Unknown Or Invalid Assumed Zero
(未知的或无效的常量标识符, 假定为零)
跟在元命令之后的一个常量标识符 (如 \$DEBUG:A) 是未知的或类型不正确。
- 111 (UNASSIGNED) 未指定
- 112 Invalid Numeric Constant Assumed Zero
(无效的数值常量, 假定为零)
跟在元命令之后的数值常量 (如: \$debug:123456) 的格式错误或数值超出范围。
- 113 Invalid Meta Value Assumed Zero
(无变量的值无效, 假定为零)
跟在元命令之后的既不是常量也不是标识符, 假定其值为零。
- 114 Invalid Meta Command (无效的元命令)
元命令之后要有+, -或:, 但未找到, 该元命令无效。
- 115 Wrong Type Value For Meta Command Skipped
(元变量的值类型错误, 跳过该元命令)
元变量要求是个字符串, 但给出的是整数值, 或者反之。
- 116 Meta Value Out Of Range Skipped
(元变量的值超出范围, 跳过该元命令)
给 \$Linesize 的值 <16 或 >160, 给 \$real 的值不是 4 或 8, 给 \$integer 的值不是 2, 都属于这种情况。
- 117 File Identifier Too Long Skipped (文件标识符太长, 跳过)
给出在 \$include 的文件名的字符串太长, 应 ≤ 96。
- 118 Too Many File Levels (文件层次太多)
用 \$INCLUDE 给出的文件嵌套层次太多, 不处理该 \$include。
- 119 Invalid Initialize Mete (执行该元命令无效)
在 \$POP 元命令之前没有对应的 \$PUSH 元命令。
- 120 CONST IDENTIFIER EXPECTED (期望一个常量标识符)
在 \$INCONST 元命令后没有找到标识符, 不处理该元命令。

- 121 Invalid INPUT Number Assumed Zero (无效的输入值, 取为零)
用户为 \$ INCONST 输入的值是无效的。
- 122 Invalid Meta Command Skipped (无效的元命令, 跳过)
在 \$ IF 元命令的后面没跟有 \$ THEN 或 \$ ELSE。
- 123 Unexpected Meta Command Skipped (不希望的元命令, 跳过)
\$ THEN 没有对应的 \$ IF 元命令
- 124 Unexpected Meta Command (不希望的元命令)
元命令未在注释中给出; 还是被处理了。
- 125 Reserved (保留)
- 126 Invalid Real Constant (无效的实型常量)
实数类型的常量以小数点开头或以小数点结尾, 它的值还接收了。
- 127 Invalid Character Skipped (无效的字符, 跳过)
源文件的字符在程序文本中是不可接受的。
- 128 Forward Proc Missing (缺提前调用的过程)
程序中用 FORWARD 说明过提前调用的过程或函数, 但没找到这些过程和函数本身, 该出错信息出现在清单文件的符号表中。
- 129 Label Not Encountered (没有遇到标号)
找不到在标号说明中说明的标号,
该出错信息出现在清单文件的符号表中。
- 130 Program Parameter Bad (错误的程序参数)
程序参数没有被说明或者它的类型是不可接受的。
该出错信息出现在清单文件的符号表中。
- 131 (unassigned) 未指定
- 132 (unassigned) 未指定
有几种原因会出现 OVERFLOW ERRORS (溢出错误)
- 133 Type Size Overflow (类型长度溢出)
说明的数据类型隐含着有多于 65534 个字节的结构。
- 134 Constant Memory Overflow (内存常量存储区溢出)
常量存储区分配超过 65534 个字节。
- 135 Static Memory Overflow
(静态变量的内存存储区分配超过 65534 个字节。
- 136 Stack Memory OVERFLOW (堆栈存储区溢出)
堆栈的内存框区分配超过 65534 个字节。
- 137 Integer Constant Overflow (整型常量溢出)
带符号的常量表达式或整型变量的值超出范围。
- 138 Word Constant Overflow (字常量溢出)
字类型变量或不带符号的常量表达式的值超出范围。
- 139 Value Not In Range For Record (记录的值不在范围内)
记录的变体标志域的值不在变体范围内, 这在使用结构常量, 使用长格式的 NEW, DISPOSE, SIZEOF 过程或其它应用中可能遇到。

140 Too Many Compiler Labels (编译程序的标号太多)

编译程序要使用内部标号并且用户程序太大, 应把程序分为几个小的程序段。

141 Compiler (编译程序)

142 Too Many Identifier Levels (标识符层次太多)

标识符的所属层次大于15 (危险错误)

143 Compiler (编译程序)

144 Compiler (编译程序)

145 Identifier Already Declared (标识符已经说明过)

在给定的层次中, 同一标识符只能说明一次。

146 Unexpected End Of File (不期望的文件结束)

在语法分析过程中, 在不该结束的地方遇到源文件结束。

常常在用户程序中出现用另外的符号代替该用的正确符号的错误, 编译程序将改正它并继续编译, 只用警告信息向用户提示, 这些情况包括:

147 ; Assumed == (Assumed == 被认为是)

148 = Assumed :

149 := Assumed :

150 = Assumed :

151 [Assumed :

152 (Assumed :

153) Assumed :

154] Assumed)

155 , Assumed :

156 , Assumed ,

157 (unassigned) 未指定

158 (unassigned) 未指定

159 (unassigned) 未指定

160 (unassigned) 未指定

161 (unassigned) 未指定

如果在源程序中要求有特定的符号, 但找不到, 编译程序可能以自动插入它并给出如下的警告信息:

163 Insert ,

164 Insert ;

165 Insert -

166 Insert :=

167 Insert OF

168 Insert)

169 Insert)

170 Insert :

171 Insert (

172 Insert DO

173 Insert
 174 Insert .
 175 Insert .
 176 Insert END
 177 Insert TO
 178 Insert THEN
 179 Insert *
 181 (unassigned) 未指定
 182 (unassigned) 未指定
 183 (unassigned) 未指定
 184 (unassigned) 未指定

如果源程序中所希望的特定符号是某些无效的符号后面找到的，那么无效的符号被跳过，并给出如下警告信息：

185 Invalid Symbol Begin Skip (开始跳过无效的符号)
 186 End Skip (跳到此处结束)
 187 End Skip (跳到此处结束)

这一跳结束用于由短语“Begin Skip”开始的跳过源文本的非法字符的情况。186 只用于185。

188 Section Or Expression Too Long (部分或表达式太长)
 编译程序在分析表达式时已无法继续下去，试图重新安排程序，或用对中间值进行赋值来分解表达式。

190 Invalid Real Operator Or Function
 (无效的实型运算符或函数)
 例如，对实型量使用MOD运算符，或使用ODD函数。

191 Invalid Value Type For Operator Or Function
 (给运算符或函数的值的类型是无效的)
 例如，对枚举类型的数据进行MOD运算或使用ODD函数。

192 (unassigned) 未指定

193 (unassigned) 未指定

194 Type Too Long (类型太长)
 使用大于32767个字节的变量或类型。

195 Compiler (编译程序)

196 Zero Size Value (零长度值)
 使用空记录“RECORD END”作为长度。

197 Compiler (译程序)

198 Constant Expression Value Out Of Range
 (常量表达式的值超出范围)
 用于数组下标，子界赋值，其它的子界的值超出范围。

199 Integer Type Not Compatible With Word Type
 (整型类型与字类型不兼容)

这是把类型量与字类型量混用的一种错误，在进行算术运算时，应分清带符号和不带符号的两种数据，要么用WORD () 把带正符号的值变为不带符号的值，要么用ORD () 把不带符号的值 (< maxint >) 变为带符号的值。

- 200 (unassigned) 未指定
- 201 Type Not Assignment Compathible (类型不赋值兼容)
赋值语句或数值参数用的类型不兼容。
- 202 Type Not Compatible In Expression (表达式中类型不兼容)
表达式中混用了不兼容的类型。
- 203 Not Array Begin Skip (非数组开始跳过)
非数组变量后是一个左方括号 (或圆括号)，编译程序将跳过它直到给出187号出错信息处。
- 204 Invalid Ordinal Expression Assumed Integer Zero
(无效的有序表达式，假定为整型零)
表达式的类型错误或是非有序类型。
- 205 Invalid Use of PACKED Components
(无效的使用PACKED分量)
PACKED结构的分量没有地址 (它可能不在字节的边界上)，并且它不能通过引用来传送。
- 206 Not Record Field Ignored (不是记录的域，不予处理)
变量后面跟一个点，但它又不是记录，地址，或文件。
- 207 Invalid Field (无效的域)
记录变量和点之后没跟有效的域名。
- 208 File Dereference Considered Harmful
(间接引用文件被看作为有害的)
在计算文件缓冲变量的地址时，不能做那些要用缓冲变量来做的特定处理，(就是文本文件的延迟求值或二进制文件的并发操作)，因此，在该地址的缓冲变量可能是无效的。实践证明该用法可能是有害的。
- 209 Cannot Dereference Value (不能被引用的值)
或面跟有箭头的变量不是指针，地址或文件，编译程序不能引用它指的值。
- 210 Invalid Segment Address (无效的段地址)
变量要用段地址，但需要一个默认的段地址，可能需要进行变量的局部复制
- 211 Ordinal Expression Invalid Or Not Constant
(有序表达式无效或不是常量)
此处需要一个常量有序表达式。
- 212 (unassigned) 未指定
- 213 (unassigned) 未指定
- 214 Out of Range For Set 255 Assumed
(超出规定的255的集合范围)
集合常量的元素必须有小于等于255的有序值。
- 215 Type Too Long Or Constant File begin Skip

(类型太长或包含文件开始跳过)

遇到的结构常量, 长度超出255个字节, 或者它还含有文件类型或 LSTRING 类型。

216 Extra Array Component Ignored (不管多出的数组元素)

数组常量含有比类型定义还多的分量, 对多出的不予处理。

217 Extra Record Components Ignored (不管多出的记录元素)

记录常量中含有比记录类型定义还多的分量。

218 Constant Value Expected Zero Assumed

(期望常量值, 假设其值为零)

结构常量中的出现非常量值。

219 (unassigned) 未指定

220 Compiler (编译程序)

221 Components Expected For Type (此类型要求更多的分量)

该类型的结构常量需要有更多个分量。

222 Overflow 255 Components In String Constant

字符串常量的长度超过255)

字符串常量只能占255个或小于255个字节。

223 Use NULL (使用NULL)

必需使用预先说明的常量NULL, 而不是两个引号。

224 Cannot Assign With Supertype Lstring

(不能用高级类型Lstring 来赋值)

高级数组Lstring不得出现在赋值语句中。

225 String Expression Not Constant (字符串表达式不是常量)

用星号串接的字符串只适用常量

226 String Expected Character 255 Assumed

(期望字符串, 假设其长度为255)

遇到一个不含字符的串常量, 可能导致使用 NULL 值, 并且假设其长度为 255。

227 Cannot Assign To Function (不能对函数赋值)

不能在函数体外向函数赋值, 此时可用RESULT过程。

228 Cannot Assign To Variable (不能对变量赋值)

不得向READONLY, CONST属性的变量或FOR语句的控制变量进行赋值

229 Cannot Use CONST Parameter Of Address Assumed Zero

(不能按照CONST参数或地址的理论值来使用)

表达式没有地址和不能是参考参数

230 Unknown Identifier Assumed Integer Begin Skip

(未知的标识符, 假设为整型, 开始跳过)

标识符需要地址, 并且要跳到或面的逗号, 分号或右括号。

231 VAR Parameter Or WITH Record Assumed Integer Begin Skip

变参或开域语句记录错误, 认为是整型量, 开始跳)

无效的标识符需要用变量地址，并且要跳到后面的逗号，分号或右括号。

232 Cannot Assign To Typo (不能赋值)

不能向文件赋值，或由于其它原因不能赋值。

233 Invalid Procedure Or Function Parameter Begin Skip

(无效的过程或函数参数，开始跳过)。

在使用内部过程和函数可能出错，原因诸如：

1. NEW或DISPOSE的第一个参数不是指针变量
2. 长格式的NEW, DISPOSE, SIZEOF 过程中无记录变体标志
3. 长格式的NEW, DISPOSE, SIZEOF 过程中的高级数组下标太多
4. 长格式的NEW, DISPOSE, SIZEOF 过程中的高级数组下标太少
5. NEW或SIZEOF过程中的高级数组没给出下标界值
6. 对不是有序类型的值进行ORD或WRD操作
7. 对无效的值或类型进行LOWER或UPPER操作
8. 对高级数组，文件，数组进行不恰当的PACK或UNPACK操作
9. RETYPE的第一个参数不是类型标识符
10. RESULT的参数不是函数标识符
11. 使用尚不可用的内部过程或函数
12. 对INTEGER4的数据用ORD或WRD求值出界
13. 把无序的或非INTEGER4的数据作为HIWORD或LOWORD的参数

234 Type Invalid Assumed Integer (类型无效，假设为整型)

READ, WRITE, ENCODE或DECODE的参数不是INTEGER, WORD, REAL, BOOLEAN, 枚举及指针类型，或者给READ或WRITE的类型不是CHAR, STRING, LSTRING, 或者给 READFN 的这三种类型，也不是FILE 类型，编译程序将认为它们是整型，这种错误也可能出现在程序参数表中，当程序参数属于不可读入的类型时，将在主程序的 BEGIN 处给出这一出错信息。

235 Assumed File Input (假设是INPUT文件)

当READFN的第一个参数不是文件时，就假定用INPUT文件

236 Invalid Segment For File

文件的参数总要在默认的数据段中。

237 Assumed Input (认为是输入)

在程序参数表中没给出INPUT，假设给了。

238 Assumed Output (认为是输出)

在程序参数表中没给出OUTPUT，假设给了。

239 Not Lining Or Invalid Segment

READSET, ENCODE或DECODE的参数必须是一个LSTRING，并且必须在默认的数据段中，不满足这个条件是错误的。

240 (unassigned) 来指定

241 Invalid Segment Variable (无效的段变量)

变量要用段地址，需要有一个默认的值，可能要对变量进行局部拷贝。

- 242 File Parameter Expelcted Begin Skip
READSET过程要有一个文本文件参数，但未找到，编译程序将不处理该过程，并认为此处出现187号错误。
- 243 Character Set EXPECTED (期望字符集合)
READSET过程要有一个文本文件参数，但未找到。
- 244 Unexpeted Parameter Begin Skip (不希望的参数，开始跳过)
在函数EOF, EOLN和PAGE中有多于一个的参数，对多出部分不予处理。
- 245 Not Text File (非文本文件)
EOLN, PAGE, READ和WRITE只适用于文本文件。
- 246 (unassigned) 未指定
- 247 Invalid Function (无效的函数)
[不能用]
- 248 Size Not Identical (长度不相等)
RETYPE不能正确地执行，因为指定的参数不等长。
- 249 Procedural Type Parameter List Not Compatible
(过程类型的参数不兼容)
过程类型的形式参数与实际参数不兼容；可能是参数的个数不同，或函数结果类型不同，或参数类型不同，或属性不同。
- 250 Cannot Use Procedure With Attribute
企图调用直接或间接带有INTERRUPT属性的过程，INTERRUPT尚未可用。
- 251 Unexpected Parameter Begin Skip (不希望参数，开始跳过)
过程或函数后给出左括号，但找不到参数，开始跳直到出现187号信息的地方。
- 252 Cannot Use Procedure Or Function As Parameter
(不能把这类过程或函数当作参数使用)
内部过程或函数不能当作参数来传递
- 253 Parameter Not Procedure Or Function Begin Skip
(参数不是过程或函数，开始跳过)
这里所要求的是过程参数，但找不到，跳直到出现187号信息处。
- 254 Supertype Array Parameter Not Compatible
(高级数组类型参数不兼容)
实际参数与形式参数类型不同，或者不是由与形式参数相同的高级类型派生出来的。
- 255 Compiler (编译程序)
- 256 VAR Or CONST Parameter Types Not Compatible
(VAR或CONST的参数类型不兼容)
地址引用的实际参数和形式参数的类型必须是等同的。
- 257 Parameter List Sise Wrong Begin Skip
(参数表长度错误，开始跳过)
参数太少或太多，在太多时跳过多出部分。

- 258 Invalid Procedural Parameter To EXTERN
(对EXTERN过程此过程参数无效)
把既不为 EXTERN 也不为 PUBLIC 的过程或函数作为参数传给说明为 EXTERN的过程或函数,这是不能处理的。可以用对过程或函数给出 PUBLIC 属性来解决。
- 259 Invalid Set Constant For Type (类型的集合常量是无效的)
集合元素值不是常数,或其基类型不等同,或者常量值太大。
- 260 Unkown Identifier In Expression Assumed Zero
(表达式中有未知的标识符,假定为零)
表达式中的标识符还没有定义过,可能是拼写错误。
- 261 Identifier Wrong In Expression Assumed Zero
(表达式中标识符错误,假定为零)
表达式中的一个标识符错误,例如,用了文件类型的标识符。
- 262 Assumed Parameter Index Or Field Begin Skip
(假设参数为数组下标或记录的域,开始跳)
在出现260或261号错误后,圆括号或方括号中的全部内容,以及标识符后的点都被跳过。
- 263 (unassigned) 未指定
- 264 (unassigned) 未指定
- 265 Invalid Numeric Constant Assumed Zero
(无效数值常量,假定为零)
是在用DECODE函数变一串字符为INTEGER或INTEGER4常量时出现的错误,如值太大,有非法字符等等。
- 266 (unassigned) 未指定
- 267 Invalid Real Numeric Constant (无效的实型常量,假定为零)
是在用DECODE函数变一串字符为REAL常量时出现的错误,如值太大,有非法字符等等,
- 268 Cannot Begin Expression Skipped (不能开始表达式,跳过)
此符号不能开始表达式分析,所以删掉。
- 269 Cannot Begin Expression Assumed Zero
(不能开始表达式,假定为零)
此符号不能开始表达式分析,用零替代。
- 270 Constant Overflow (常量溢出)
在DIV或MOD运算中除数为零,这是非法的。
- 272 Word Constant Overflow (字常量溢出)
WORD常量减去WORD常量得到负的结果。
- 273 (unassigned) 未指定
- 274 (unassigned) 未指定
- 275 Invalid Range (无效的范围)
子界的下界值大于它的上界值(例如:2..1)

- 276 CASE Constant Expected (期望CASE常量)
期望一个CASE语句的常量, 或记录变体的一个常量值, 但未找到。
- 277 Value Already in Use (该值已经使用过)
在CASE语句或记录变体中, 该值已经用过了。
(例如, CASE 1..3:XX; 2:YYY end)
- 278 Reserved (保留的)
- 279 Label Expected (希望有标号)
此处需要的一个标号但没有找到。
- 280 Invalid Integer Label (无效整型标号)
非十进制记数法 (例如, 8#77) 不允许用在标号中。
- 281 Label Assumed Declared (假定标号已经说明过)
该标号没有在LABEL部分中说明过。
- 282 (unassigned) 未指定
- 283 Expression Not Boolean Type (表达式不是非布尔类型)
IF, WHILE或UNTIL后的表达式必须是布尔类型的。
- 284 Skip To End Of Statamend (跳到语句末)
跳过不正确的ELSE或UNTIL子语句。
- 285 Compiler (编译程序)
- 286 ; Ignored (不管此;)
ELSE前面的分号总是错的, 跳过去。
- 287 (unassigned) 未指定
- 288 Skipped (跳过此;)
OTHERWISE后的分号总是错的, 跳过去。
- 289 Variable Expected For FOR Statement Begin Skip
(FOR语句中要有变量标识符, 但未找到, 开始跳)
变量标识符必须跟在FOR后面, 跳直到出现187号信息处。
- 290 (unassigned) 未指定
- 291 FOR Variable Not Ordinal Or Static Or Declared In Procedure
FOR语句的循环控制变量不是有序的, 或不是静态的, 或者是在过程中说明的)
FOR语句控制变量不应该是:
- 实数类型, 或INTEGER4类型, 或其他非有序类型
 - 数组, 记录或文件类型的分量
 - 引用的指针类型或地址类型
 - 在栈或堆中的变量, 除非局部地加以说明
 - 未作局部说明的变量, 除非在静态存储区中
 - 变量参数 (VAR或VARS参数)
 - 用ORIGIN属性给出的带段地址的变量
- 292 Skip To: = (跳到:=处)
FOR语句中此处应有赋值号, 但未找到, 跳到下个:=处。

- 293 Reserved (保留)
- 294 GOTO Considered Harmful (认为GOTO很有害)
当使用了\$GOTO元命令后, 将对所有GOTO语句给此信息。
- 295 (unassigned) 未指定
- 296 Label Not Loop Label (标号非循环标号)
在BREAK或CYCLE语句中用的标号不是循环语句的标号, 即它不是FOR, WHILE或REPEAT语句的标号。
- 297 Not In Loop (不在循环中)
BREAK或CYCLE语句不在FOR, WHILE或REPEAT语句中。
- 298 Record Expected Begin Skip (期望记录, 开始跳过)
WIHT语句中要有一个记录变量, 但找不到, 跳直到出现137号错误处。
- 299 (unassigned) 未指定
- 300 Label Already In Use Previous Use Ignord
(标号已经在前面出现过, 不管以前的使用)
该标号已经在一个语句的前面出现过。
- 301 Invalid Use Of Procedure Or Function Parameter
(无效地使用过程或函数参数)
过程参数被当作函数参数使用, 或把函数参数当过程参数使用。
- 302 (unassigned) 未指定
- 303 Unkown Identifier Skip Statement (未知的标识符, 跳过语句)
在语句开始处遇到一个没有定义过(或拼错?)的标识符, 整个语句都不处理。
- 304 Invalid Identifier Skip Statement (无效的标识符, 跳过语句)
在语句开始处遇到一个不正确的标识符(如文件类型的标识符), 整个语句都不处理。
- 305 Statement Not Expected (非希望的语句)
在一个模块或不需要初始化的IMPLEMENTATION中遇到由BEGIN和END"夹"起来的执行体。
- 307 Unexpected END Skipped (不希望的END, 跳过)
遇到一个不与任何一个BEGIN, CASE或RECORD配对的END。
- 308 Compiler (编译程序)
- 309 Attribute Invalid (无效的属性)
把只能给过程或函数的属性用于变量, 或把只能给变量的属性用于过程或函数, 或混用了相矛盾的属性, 如PUBLIC和EXTERN。
- 310 Attribute Expected (期望属性)
左方括号表示要给出属性, 但未找到。
- 311 Skip To Identifier (跳到标识符处)
跳过一个无效的该符号, 跳到跟在后面的标识符处。
- 312 Identifier Expicted (期望标识符)

起望有标识符表,但这不是一个标识符。

- 313 Reserved (保留的)
- 314 Identifier ExPected Skip To; (期望标识符,跳到 ; 处)
期望有要说明的新标识符,但没有找到,跳到下一个"; "处。
- 315 Type Unknown Or Invalid Assumed Integer Begin Skip
(类型是未知的或是无效的,假设为整型,开始跳)
参数或函数返回的数据类型不正确,这可能它不是标识符,未被说明过,或者参数或函数返回的值是文件或高级数组,跳直到给出187号出错信息处。
- 316 Identifier ExPected (期望标识符)
在参数表中的保留字PROCEDURE或FUNCTION之后要有一个标识符,但未找到。
- 317 (unassigned) 未指定
- 318 Compiler (编译程序)
- 319 Compiler(编译程序)
- 320 Previous Forward Skip Parameter List
对那些用FORWARD定义过的(或在接口中定义过的)过程或函数,不能再重复给出它们的参数表内容和函数返回的类型。
- 321 Not EXTERN(不能为EXTERN)
对带有ORIGIN属性的过程又给出了EXTERN属性。
- 322 Invalid Attribute With Function Or Parameter
(过程或函数有非法属性)
发现不正确地使用了INTERRUPT 属性的过程,可能它是一个函数或有一些参数属性错误。
- 323 Invalid Attribute In Procedure Or Function
(被嵌套的过程或函数的带有属性,或者被说明为EXTERN
这是不合法的。
- 324 Compiler(编译程序)
- 325 Already Forward(已说明为FORWARD)
对同一个过程或函数,不能使用两次FORWARD说明。
- 326 Identifier ExPected For Procedure Or Function
(过程或函数期望标识符)
关键字PROCEDURE或FUNCTION之后必须跟有标识符。
- 327 INvalid Symbol Skipped(跳过无效的符号)
在接口中决不能使用FORWARD或EXTERN伪命令。
- 328 Extern Invalid With Attribute(EXTERN带有非法属性)
EXTERN过程不能有 PUBLIC 属性。
- 329 Ordinal Type Identifier ExPected Integer Assumed Begin Skip
(期望有序的类型标识符,假设为整型,开始跳过)
记录变体标识类型应为有序类型的标识符,但找不到。
- 330 Contans File Cannot Initialize(包含文件,不能初始化)

虽然允许把文件放在记录变体中，但这是不安全的，编译程序不能自动调用 NEWFQQ 对该文件进行初始化。

- 331 Type Identifier Expected Assumed Character
(期望类型标识符，假设为字符)
期望一个有序的类型标识符，但未找到，假设它为字符类型标识符。
- 332 Reserved(保留)
- 333 Not Supertype Assumed String(非高级类型，假设为字符串)
看起来貌似高级数组类型的派生符，但类型标识符又不是高级数组类型，就认为它是 STRING 类型。
- 334 Type Expected Integer Assumed(期望类型，假定为整型)
期望有类型子句或类型标识符，但找不到，假设它为整型标识符。
- 335 Out Of Range 255 For LSTRING(LSTRING 超出了 255 范围)
LSTRING 派生符不能有大于 255 的上界。
- 336 Cannot Use Supertype Use Designator
(不能使用高级类型，要使用它的派生符)
高级数组类型必须是变量形式参数或指针引用，不能用它说明另外的变量。
- 337 Supertype Designator Not Found(高级类型派生符没找到)
期望一个高级数组的派生符，但未找到，它给了高级数组的上界。
- 338 Contains File Cannot Initialize(包含有文件，不能初始化)
虽然允许使用高级数组的文件，但这是不安全的，编译程序不能自动调用 NEWFQQ 对该文件进行初始化。
- 339 Supertype Not Array Skip To; Assumed Integer
(高级类型不是数组，跳到 ; 处，假设为整型)
在类型子句中，SUPER 之后总是要跟有关键字 ARRAY，但未找到，假设为整型，跳到下一个分号处。
- 340 Invalid Set Range Integer Zero To 255 Assumed
(无效的集合范围，假定为整数 0..255)
一个集合的基本类型必须在 0 到 255 的子界范围之内。
- 341 File Contains File(文件包含有文件)
文件类型不能直接地或间接地包含有文件类型
- 342 PACKED Identifier Invalid Ignored
(无效的 PACKED 标识符)
关键字 PACKED 之后必须跟着 ARRAY, RECORD, SET 或 FILE，跟有任何其它类型的标识符都是非法的。
- 343 Unexpected PACKED(不期望的 PACKED)
关键字 PACKED 只能用于结构类型(见上条)。
- 345 Skip To; (跳到 ;)
分号应出现在说明的末尾，但找不到，不能用行结束代替，假设下一个分号是说明的结束位置。
- 346 Insert; (插入 ;)

分号应出现在说明的末尾，但找不到，假设在此处插入了一个分号。

- 347 Cannot Use Value Section With ROM Memory
用了元命令 \$ROM，就不能再使用VALUE设置变量初值。
- 348 UNIT Procedure Or Function Invalid EXTERN
(UNIT中的过程或函数无效的EXTERN)
在接口中，不能说明过程和函数为EXTERN，必须到IMPLEMENTATION的开头处说明。
- 349 (unassigned) 未指定
- 350 Not Array Begin Skip (非数组，开始跳过)
在VALUE段中的后面跟有方括号的变量不是数组。
- 351 Not Record Begin Skip (非记录，开始跳过)
在VALUE段中的后面跟有字符“.”的变量不是记录。
- 352 Invalid Field (无效的域)
在VALUE段中的被看作是域的标识符不记录中。
- 353 Constant Value Expected (期望是常量值)
在VALUE部分中，变量只能被初始化成常量值。
- 354 Not Assignment Operator Skip To; (非赋值运算符，跳到，处)
在VALUE部分中，遗漏了赋值运算符。
- 355 Cannot Initialize Identifier Skip To;
(不能初始化标识符，跳到，处)
在VALUE部分中的符号，不是在该层次所说明的静态变量，或它具有非法的ORIGIN或EXTERN属性。
- 356 Cannot Use Value Section (不能使用VALUE部分)
VALUE部分不能用在INTERFACE部分，要用在IMPLEMENTATION中。
- 357 Unknown Forward Pointer Type Assumed Integer
(未知的超前引用指针类型，假定为整型)
在前面说明引用类型时用到的一个类型标识符本身没在TYPE (或者VAR) 部分中说明。
- 358 Pointer Type Assumed Forward (假定指针类型是超前引用的)
在TYPE部分，在出现引用指针或地址类型的情况时，若被引用的类型已经在前面的有效作用范围内说明过，而在同一TYPE段中稍后面的部分再次对该类型进行新的说明，编译程序将使用这后一次的定义。
例如：

```
PROGRAM OUTSIDE;  
TYPE A = WORD;  
PROCEDURE P;  
  TYPE C = ^A;  
  A = WORD;
```

这时将给出Pointer Type Assumed Forward的提示信息。

- 359 Cannot Use Label Section (不能使用标号部分)
LABEL部分不能用在INTERFACE部分，要用在IMPLEMENTATION中。
- 360 Forward Pointer To Supertype
在TYPE部分说明引用的引用类型是高级数组，但在引用时尚未给出对这一高级数组的说明。
- 361 Constant Expression Expected Zero Assumed
(期望常量表达式，假定为零)
在CONST部分中的表达式不是常量。
- 362 Attribute Invalid (属性无效)
在VAR部分中，错误地把EXTERN和PUBLIC或ORIGIN混用了，或者把ORIGIN给出在关键字VAR之后的方括号中。
- 363 (unassigned) 未指定
- 364 Contains File Initialize Module (包含有文件初始化的模块)
遇到一个尚无初始化过的在模块中的文件变量。当文件变量在一个模块中说明时，必须把模块作为无参数的过程调用一次，以便初始化这些文件。
- 365 * Origin Variable Contains File Cannot Initialize
遇到一个带有ORIGIN属性的文件变量，因为ORIGIN属性的变量从来不自动被初始化，必须由用户本身来初始化该文件变量。
- 366 UNIT Identifier Expected Skip To:
(期望UNIT的标识符，跳到；)
在USES之后要跟有一个标识符，但找不到。
- 367 Initialize MODULE to Initialize UNIT
(初始化MODULE来初始化UNIT)
USES子语句要执行一次群初始化调用，用户必须把模块作为一个无参数的过程调用一次，以便执行一次模块初始化操作。
- 368 Identifier List Too Long Extra Assumed Integer
(标识符表超长，假定为整型)
在带有标识符表的USES子句中，在表中所找到的标识符多于接口中给出的要素数，假设多出的标识符是与INTEGER等同的类型标识符。
- 369 End of UNIT Identifier List Ignored
在带有标识符表的USES子句中，在表中所找到的标识符少于接口的要素数，在接口中的剩余的要素将不作为USES子语句中的一部分。
- 370 (unassigned) 未指定
- 371 UNIT Identifier Expected (期望UNIT的标识符)
在"INTERFACE; UNIT"短语之后找不到UNIT的标识符。
- 372 Compiler (编译程序)
- 373 Identifier In UNIT List Not Declared
(在UNIT表中的标识符没有说明过)
在接口的UNIT表中，有一个标识符没有在接口体中进行说明。
- 374 Program Identifier Expected (期望程序标识符)

在关键字PROGRAM或MODULE之后没有标识符（致命的错误）。

375 UNIT Identifier Expected（期望UNIT的标识符）

在短语“IMPLEMENTATION OF”之后没有群标识符（致命的错误）

376 Program Not Found（没有找到程序）

没有找到关键字PROGRAM, MODULE或者IMPLEMENTATION OF,

（致命的错误）。如果源文件不是PASCAL的编译对象，就可能出现这种问题。

377 File End Expected Skip To End（期望结束文件，跳到文件末）

已经遇到了编译对象应该结束的条件，不再处理这之后的任何内容，直接进到文件末尾。

378 Program Not Found（没找到程序）

编译程序找不到被编译对象的执行体，或者未找到执行体的正确结尾。

2. PAS2.EXE发现的错误

PAS2.EXE会发现两种错误：用户错误和内部错误。用户错误很少见。所有的错误都与PAS1.EXE检测不到的那些缺陷有关。

大量的内部一致性检查都是由PAS2.EXE完成的，正常情况下，这些检查的结果都应该是正确无误的，不应该给出内部错误。如果发现了用户错误或内部错误，编译程序将立即停止编译，并给出一个出错编号和错误在源程序列表文件中的大约的行号值。

用户错误

(1) Attempt to divide by zero

零作除数，例如：A DIV 0

(2) Overflow during integer constant folding

整型常量合并时产生溢出，例如MAXINT + A + MAXINT

(3) Expression too complex/Too many internal label

表达式太复杂/内部标号太多

应设法把太复杂的表达式分解成几个表达式，并用它们的中间结果进一步产生最终结果。

内部错误

这些错误有如下格式：

*** Internal Error NNN

NNN是内部编号，范围从1到999。如果遇到这种内部错误，很难再做什么事情。可以把出错情况报告给计算机厂家求得答复，或试着修改一下出错行附近的程序内容。

运行系统发现的错误

在程序运行期间检查出的错误，被分成文件系统错误和其它种类错误两大类。文件系统错误编码的范围从1000到1999，其中编码1000到1099是操作系统的错误（来自UNIT U），1100到1199是PASCAL的文件系统的错误（来自UNIT F）。

文件系统错误

文件系统的错误有着如下的格式：

错误类型 error in 文件名

后面跟一个错误编码

错误类型字段反映的是文件控制表中ERRS域的内容。在括号中的字母表明是哪一个UNIT (U和F) 产生的这个错误。错误编码给出如下:

编码	错误性质
0	无错
1	保留
2	保留
3	操作错误 (U、F)
4	保留
5	保留
6	保留
7	文件名错 (U、F)
8	磁盘满 (U)
9	保留
10	文件找不到 (U)
11	保留
12	保留
13	文件未打开 (F)
14	日期格式错 (F)
15	行太长 (U、F)

UNIT U 错误

- 1000 Write Error When Closing File
(关闭文件时写操作错误)
- 1001 Unkown Device Name
(非法外设名)
- 1002 File Ename Extension With More Than 3 Character
(文件扩展名多于3个字符)
- 1003 Error During Creation Of New File
(建立新文件时出错)
- 1004 Error During Opening Of Existing File
(打开现存文件时出错)
- 1005 Filename With More Than 8 Or Zero Characters
(文件名多于8个字符或没字符)
- 1006 Reserved
(保留)
- 1007 Total Filename Length Over 12 Characters
(文件名的总长度超过12个字符)
- 1008 Writing Error When Advancing To Next Record
(前移到下一记录时写操作出错)
- 1009 File Too Big (Over 65535 Logical Sectors)

(文件太大(超过65535个逻辑扇区))

1010 Write Error When Seeking To Direct Record

(寻找直接记录时写操作错)

PASCAL文件系统错误代码

1100 Assign Or Readln Of Filename To Open File

(给打开文件进行assign或readln of文件名)

1101 Reference To Buffer Variable Of Closed File

(对关闭文件的缓冲变量进行引用)

1102 Textfile Read Or Wrote Call To Closed File

(对关闭文件进行文本文件的READ或WRITE调用)

1103 Read When Eof Is True (Sequential Mode)

(当EOF是真时,进行READ)(顺序方式)

1104 Read To Rewrite File, Or Write To Reset File (Sequential Mode)

(REWRITE的文件进行READ,或对RESET的文件进行WRITE(顺序方式))

1105 Eof Call To Closed File

(对关闭的文件进行EOF调用)

1106 Get Call To Closed File

(对关闭的文件进行GET调用)

1107 GET Call When EOF的 is True (Sequential Mode)

(当EOF为真时,进行GET调用(顺序方式))

1108 GET Call To Rewrite File (Sequential Mode)

(对REWRITE的文件进行GET调用(顺序方式))

1109 Put Call To Closed File (Sequential Mode)

(对RESET的文件进行PUT调用(顺序方式))

1110 Put Call To Reset File (Sequential Mode)

(对RESET的文件进行PUT调用(顺序方式))

1111 Line Too Long In Direct Textfile

(DIRECT文本文件中太长)

1112 Decode Error In Textfile Read Boolean

(在文本文件READ BOOLEAN时译码出错)

1113 Value Out Of Range In Textfile Read Char

(在文本文件READ INTEGER时读值超出范围)

1114 Decode Error In Textfile Read Integer

(在文本文件READ INTEGER时译码出错)

1115 Decode Error In Textfile Read Sint (Integer Subrange)

(在文本文件READ SINT时译码出错(整型子界))

1116 Decode Error In Textfile Read Real

(在文本文件READ时译码出错)

- 1117 Lstring Target Not Big Enough In Readset
(在READSET时LSTRING的目标不够大)
- 1118 Decode Error In Textfile Read Word
(在文本文件READ WORD中译码出错)
- 1119 Decode Error In Textfile Read Byte
(在文本文件READ BYTE时译码出错)
- 1120 Seek Call To Closed File
(对关闭的文件进行SEEK调用)
- 1121 Seek Call To File Not In Direct Mode
(对不处在DIRECT方式下的文件进行SEEK调用)
- 1122 Encode Error (Field Width>255) In Textfile Write Real
(在文本文件进行WRITE REAL时编码出错 (字段宽度大于255))

其它运行错误

非文件系统错误码的范围为2000到2999, 在某些情况下, 元命令控制着是否要进行错误检验, 在其他情况下, 总要进行错误检验, 若有控制检验错误检查的元命令, 则用下表加以说明:

2000.2049 内存错误

因为栈与堆是相对生长的, 所以会出现这种错误, 例如, 不使用 \$STACK+ 栈溢出就可以引起“栈区无效”的错误。

2000 Overflow

(溢出)

当调用一个过程或函数时, 栈(结构)溢出可用内存区, 检查是否 \$stack+ 或其他情况。

2001 No Room In Heap

(堆中无空间)

当处在NEW (GETHQQ) 过程时, 堆中已没有足够的空间供新变量使用, 总要进行捕获。

2002 HEAP IS INVALID

(堆栈无效)

当处在NEW(GETHQQ) 过程中时, 发现堆结构的分配法是错误的, 总要进行捕获。

2003 Reserved

(保留的)

2031 Nil Pointer Reference

(NIL 指针引用)

DISPOSE或 \$NILCK+发现引用一个带有NIL值的指针

2032 Uninitialized Pointer

(未初始化的指针)

DISPOSE或 \$NILCK, 发现有未初始化 (值为1) 的指针, 如果使用 \$initck, 指针只能取得此值。

2033 Invalid Pointer Range

(无效的指针范围)

DISPOSE或\$NILCK+发现有不是指向堆的指针,即是无效的指针,有可能是指向从堆中删去的DISPOSE块和返回给系统的指针。

2034 Pointer To Disposed Var

(指向已解除的VAR的指针)

(指向已解除的VAR的指针)

DISPOSE或\$NILCK+发现有指向已经取消的堆块指针,对同一个变量调用两次DISPOSE是无效的。

2035 Long Disposed Size Unequal

(长格式DISPOSE的长度不等)

当使用长格式的DISPOSE时,变量的实际长度不等于已知标记值的长度

2050..2099有序算术送算

2050 No Case Value Matches Selector

(没有CASE值选择符配对)

在没有OTHERWISE子句的CASE语句中,没有一个分支语句含有与选择符表达式值相等的标号,要检验是否使用了\$RANGECK。

2051 Unsigned Divide By Zero

(不带符号被零除)

WORD被零除,要检验是否使用\$MATHCK+。

2052 Signed Divede By Zero

(带符号被零除)

INTEGER值被零除,要检验是否使用了\$MATHCK+。

2053 Unsigned Math Overflow

(有带符号的数学溢出)

WORD结果不在0..MAXIN范围之内,要检验是否使用了\$MATHCK+。

2054 Signed Math Overflow

(带符号的数学溢出)

INTEGER的值不在-MAXINT..+MAXIN范围之内,要检验是否使用了\$MATHCK+。

2055 Unsigned Value Out Of Range

(无符号数的值超出范围)

赋值或数值参数中的源操作数的值超出了目的操作数允许值的范围,后者可以是WORD(包括BYTE)的子界,或CHAR,或枚举类型,此时可能出现由SUCC或PRED引起的值越界,或在指定一个LSTRING的长度时出现值越界,可用\$RANGECK+进行检查,当数组使用不带符号的下标并出现下标越界时也产生这种信息,这可用\$INDEXCK+来检查。

2056 Signed Value Out of Range

(带符号的值超出范围)

同上,但适用于INTEGER类型及其子界

- 2057 Uninitialized 16 Bit Integer Used
(16位整数未初始化)
- 2058 Uninitialized 16 Bit Integer Used
(8位整数未初始化)
- 2084 Integer Zero To Negative Power
(对零求负(的指数的错误))
- 2100..2149类型REAL算术运算**
- 2100 Realdive By Zero
(REAL被零除)
REAL值被零除；总要进行检查。
- 2101 Real Math Overflow
(REAL算术运算溢出)
表达式的REAL值太大；总要进行检查
- 2102 Sin Or Cos Argeument Range
(SIN或者COS的自变量范围错)
SIN或COS函数的自变量太大了，以致不能得出有意义的结果
- 2103 Exp Argument Range
(EXP 自变量范围错)
EXP函数的自变量太大了，以致不能把结果放入表达式中。
- 2104 SQRT Of Negative Argument
(求负数的平方根)
平方根函数的自变量小于零；总要捕获。
- 2105 Ln of Non-Positive Argument
(计算非正数的自然对数)
自然对数的自变量小于等于零；总要捕获。
- 2106 Trunc/Round Argument Range
(TRUNC/ROUND自变量范围错)
转换的REAL值超出INTEGER范围；总要捕获。
- 2131 Tangent Argument Too Small
(正切自变量太小)
TANRQQ函数的自变量太小，以致于结果无效；总要捕获。
- 2132 Arcsion Or Arccos of REAL>1.0
(Real>1.0的反正弦或反余弦)
ASNRQQ或ACSRQQ的自变量大于1；总要捕获
- 2133 Negalive Real To Real Fower
(负实数的实数幂)
RSRRQQ函数的第一个自变量小于零；总要捕获
- 2150..2199结构类型错误**
- 2150 String Toolong In Copyst
(COPYSTR中的字符串太长)

COPYSTR内部的源字符串太大了，以致于不能传给目标字符串，总要捕获

2151 Lstring Too Long In Intrinsic Procedure

(在内部过程中的LSTRING太长)

在INSERT, DELETE, CONCAT, 或COPYSTR内部过程的目的

LSTRING 太小，总要捕获。

2180 Set Element Greater Than 255

(集合的元素大于255)

集合的元素数大于最大值；总要捕获。

2181 Set Element Out of Range

(集合元素超出范围)

集合赋值中的数值参数太大了，以致于不能供目标集合用，使用了\$RAN

ECK检验之。

2200..2999其它错误

2201 Long Integer Math Overflow

(数字整数溢出)

2234 Long Integer Zero To Negative Power

(双精度整数用负指数)

2400 Illegal Code

(非法代码)

2450 Unit Version Number Mismatch

(群的文本号不符合)

在UNIT初始化期间，发现使用USES的用户和接口的实现所用的接口

文本号不相符；总要捕获。